TURING 图灵程序设计丛书

Apress®

Troubleshooting Oracle Performance, 2E

Oracle 性能诊断艺术

（第2版）

【瑞士】Christian Antognini 著
王作佳 刘迪 译

- Oracle数据库优化的里程碑式著作
  帮你系统地发现并解决Oracle数据库性能问题
- 源自一线Oracle性能优化实践
  涵盖目前所有可用版本


中国工信出版集团


人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

Troubleshooting Oracle Performance, 2E

Oracle性能诊断艺术

（第2版）

【瑞士】Christian Antognini 著
王作佳 刘迪 译

人民邮电出版社
北京

图书在版编目（CIP）数据

Oracle性能诊断艺术 ：第2版 / （瑞士）安托尼尼（Antognini,C.）著 ；王作佳，刘迪译. -- 北京 ：人民邮电出版社，2016.6
（图灵程序设计丛书）
ISBN 978-7-115-42117-3

Ⅰ. ①O… Ⅱ. ①安… ②王… ③刘… Ⅲ. ①关系数据库系统 Ⅳ. ①TP311.138

中国版本图书馆CIP数据核字(2016)第070590号

内 容 提 要

本书是兼具技术性与指导性的参考手册。书中首先介绍了全书所需的基础知识；接着描述如何借助相关工具识别和分析性能问题，如何利用动态性能视图；接着重点关注负责将 SQL 语句生成执行计划的组件——查询优化器；最后则展示了 Oracle 数据库为高效执行 SQL 语句提供的特性。

本书适合性能分析人员、Oracle 数据库应用程序开发人员与数据库管理员阅读参考。

◆ 著 [瑞士] Christian Antognini
译 王作佳 刘 迪
责任编辑 朱 巍
执行编辑 贺子娟
责任印制 彭志环
◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷
◆ 开本：800×1000 1/16
印张：39.25
字数：1020千字 2016年6月第1版
印数：1 - 3 500册 2016年6月北京第1次印刷
著作权合同登记号 图字：01-2014-6028号

定价：119.00元

读者服务热线：(010)51095186转600 印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第 8052 号

版权声明

译 者 序

一次偶然的机会，在浏览图灵网站新书的时候，无意间发现TOP这本书的第二版在招募译者。之前国内曾引进此书，作为Oracle性能调优领域的里程碑式著作，这本书给了国内DBA许多的启发。因此发现此书的第二版之后，当即决定了翻译意向，随后在与编辑联系并试译通过以后，即开始了翻译工作。此书原版共700余页，我在开始翻译之后马上就感觉到了压力，所以就联系了同为DBA的朋友、本人进入Oracle领域的引路人刘迪，请他帮忙分担一部分翻译工作。

此书从Oracle调优基础讲起，介绍了如何定位性能问题，同时对查询优化器的工作原理进行了详细描述，最后总结了一些常见的调优技术。作者对Oracle调优技术的细节把控方面令译者深感敬佩，其严谨的态度也是译者以及广大DBA从业者学习的榜样。在此，译者感谢原著作者的辛苦付出。

此书第1、2、6、7、8、9、10、14、15、16章以及文前部分由王作佳翻译，第3、4、5、11、12、13章由刘迪翻译。

此书在翻译过程中有很多名词术语，译者尽量全部翻译，遇到表达不准的术语时，均尽力采用网络上常见的翻译，另外多数不常见的术语译者都标注了原文以供读者参考。此书为译者第一部译作，因译者水平有限以及书中涉及技术较深，再加之译者时间有限，难免有误译漏译现象，还请读者见谅。如有发现错误，请通过译者邮箱或图灵网站联系以便修正。

在此感谢图灵公司的编辑朱巍老师，她给了我许多指导和帮助。同时感谢图灵其他编辑老师为本书付出的辛苦努力。在翻译初期，我的同事史盈盈女士给出了许多宝贵的建议，在此表示感谢。另外，感谢数据库组的同事们在翻译期间给予的理解和帮助。

王作佳

感谢我的团队在翻译期间给予的理解与支持。感谢王作佳提供的这次翻译机会，让我受益良多。感谢妻子孙婷的照顾与理解，能让我有时间专心翻译。感谢图灵的各位编辑对本书付出的努力。

刘迪

第二版序——Jonathan Lewis

在为本书写序言的时候，我在阅读完样章后，做的第一件事就是查看我为第一版写的序言，看看其中有多少内容需要改动。显然，参考的章节号是需要修改的。但令我吃惊的是，在关于为什么有志向的Oracle专业人士都应该读读这本书的问题上，有几个重要的观点我没能讲清楚。借此修订序言之机，我进一步阐述如下。

互联网上充斥着关于Oracle的众多信息，但是这些信息是高度碎片化的，亟需整理和提炼。许多已出版的Oracle书籍也都存在同样的问题：书中提供了大量的信息，但没有按照任何形式的逻辑体系进行讲述，这使得读者很难抓住一个主题，也就无法作为后续学习和理解的切入点。甚至，Oracle官方手册也存在同样的问题，但情况相对较好一些。在性能诊断的演示中我经常阐述的观点是，在你的阅读清单中应该包含以下三本Oracle官方手册：*Oracle Database Concepts* 手册、*Oracle Database Administrator's Guide*和*Oracle Database Performance Tuning Guide*。然而，在阅读任意一本手册时，你会发现，其中一些知识直到读完其他两本之后才能真正理解。本书的一大特色正是它在组织信息的方式上避免了上述历史问题，明确告诉我们需要达成的目的是什么，为什么要达成这些目的，以及如何达成这些目的。

有时，这种结构简单得令人难以置信。我就被书中相邻三章的标题所吸引，且不说这些章节的内容非常值得阅读，仅就标题而言，已经对某些概念进行了异常清晰的阐述，将其长久未被认识到的重要性凸显出来，它们正是在性能诊断过程中应当首先注意的问题：

- 第3章 分析可重现问题
- 第4章 实时分析不可重现问题
- 第5章 事后分析不可重现问题

你是否意识到问题的类型只有三种，而你解决问题的策略往往又取决于这三类中的哪一类？在所有的案例中，用来解决问题的数据的基本来源都是一样的，但随着时间的推移，某些数据的可用性和粒度会发生变化。因此，理解这种问题分类是系统化解决问题的第一步。

整本书都以一种相同的架构进行论述：整理各种信息，并展示各种可能性以及如何获取相应结果。例如，第6章列举了一长串Oracle优化查询时可能执行的转换，第13章则给出了一个很长的列表，展示可能出现在执行计划中基于分区操作的不同方法。

待读完本书后，你可能会发现，学到的知识比想象的要多很多；而对于本来已经知晓的知识，由于分散的知识点被整合到了一起，空白点得以补充完善，如今又有了更深入的理解。Christian的知识和见解已然让信息重构了！

——Jonathan Lewis
世界级Oracle专家，《Oracle核心技术》作者

第二版序——Cary Millsap

在过去十年间，我认为在Oracle性能领域最令人欣慰的情形是：如今，在书店买到的书籍中所承载的信息质量有了根本性的改善。

从前能买到的关于Oracle性能方面的书籍几乎如出一辙。这些书不是在暗示你的Oracle系统必然承载了太多的I/O（事实上并不一定），就是提到没有足够的内存（就像上一种说辞一样，也不是真实情况）。它们可能会堆砌罗列大量你可能会运行的SQL语句，并让你优化这些SQL，声称这样可以解决一切性能问题。

那是一个黑暗的时代。

Chris的这本书就是刺破这黑暗的光明使者之一。黑暗和光明的差别可归结为一种简单的理念，一种从你10岁起数学老师就让你反复实践的理念：演示你的做法。

我的意思并非“展示并介绍”，就像有人声称让一个拥有数百用户的站点提升了百分之几百的性能（原话就是这样说的），然后自封为专家。我所说的演示你的做法，是指先记录一个相关的基线测量，再进行一次受控实验，记录另一个基线测量，然后公开透明地公布你的结果，让读者可以跟随你的思路甚至在需要时重现你的案例。

这一点很重要。当作者们开始那样做时，Oracle爱好者们就会受益匪浅。从2000年开始，在Oracle社区中提出深层次性能问题并寻求高品质答案的人明显增加了许多。这也使得人们更迅速地剔除那些曾被许多人信服的错误方法。

本书中，Chris遵循了有效的模式。他向你讲述有用的技术。但不止于此，他还介绍自己是如何知道这些技术的，换句话讲，他告诉你如何自己找出问题答案。Chris演示了他的做法。

这样做有两个益处。首先，能让你更深入地理解他所演示的内容，从而更容易记住和应用他的课程。其次，通过理解这些例子，你不仅可以理解Chris正在演示给你的内容，同时还能够解决Chris没有提及的其他有意思的问题，比如像本书付印后Oracle的下一个版本会出现哪些新特性。

对我而言，这本书是兼具技术性与指导性的参考手册，它包含了大量文档化的可重用的作业案例。本书也包含几个有说服力的新论据，使我可以分享Chris的观点和热忱。Chris在此书中使用的论据可以帮我说服更多的人正确地做事。

Chris不仅睿智而且精力充沛，他站在了一些Oracle专家的肩上，这些人包括：Dave Ensor、Lex de Haan、Anjo Kolk、Steve Adams、Jonathan Lewis、Tom Kyte，等等。这些人都是我心中的英雄，正是他们为这个领域带来严谨之风。现在，我们也可以站在Chris的肩上了。

——Cary Millsap

Method R公司首席执行官，博客地址http://carymillsap.blogspot.com。

第一版序

我从20多年前开始使用Oracle数据库软件，大概花了3年时间才发现，在人们看来，问题诊断和调优简直神秘得不可思议。

曾经有个开发人员发给DBA团队一条性能不好的查询语句。我检查了执行计划和数据样本，然后指出大部分的工作量可以通过给其中的一张表添加一条索引来消除。开发人员的回答是："这是张小表，并不需要索引啊。"（当时是6.0.36版本的时代，顺便提一下，那时小表的定义是"不超过四个块的大小"。）最终我还是创建了索引，查询速度提升了30倍，当然我又有一大堆要解释的内容。

问题诊断并不依赖于魔法、秘诀或神话，更多的是依靠理解、观察和解释。理查德·费曼曾说过："无论你的理论有多完美，还是你有多聪明，如果你的理论和实验结论不符，那这理论就是错误的。"在Oracle性能方面有许多这样错误的"理论"，多年以前就应该从集体认知中清除掉，Christian Antognini就是一个能帮你消除错误理论的人。

在本书中，Christian着手描述事情真正的工作方式，你应该留意什么样的症状，以及这些症状代表什么含义。另外，尤其难能可贵的是，他还鼓励你要有条不紊地去进行观察与分析，并密切关注过程中出现的相关细节。有了这个建议，你就能够在出现性能问题时以最合适的方法定位出真正的症结所在。

尽管这本书很可能需要你从头至尾仔仔细细地阅读，但是不同的读者应该会以不同的方式从中获益。有些人可能偶尔在浏览时发现一些独到的见解，正如我此前多年一直搞不懂高度均衡直方图为何如此命名，而直到读了第4章后，Christian的描述才让我茅塞顿开。

一些读者会找到一些特性的简短描述帮助他们理解Oracle为什么要实现这些特性，并让他们通过案例推导与他们的应用相关联的情形。第5章关于"安全视图合并"的描述对于我来说就是这样。

另一部分读者可能会屡次重复阅读本书中的某一章节，因为这一章节含有他们正在使用的某些重要特性的许多细节。我想第9章中关于分区的深入讨论就会让人们孜孜不倦地反复阅读。

本书内容丰富，值得仔细研读。谢谢你，Christian。

——Jonathan Lewis

第二版致谢

面对现实吧，写书并不是一件值得去做的事。根本不值！写书会占用你很多的业余时间，用这些时间你本可以做些更有趣的事情。所以当我决定是否应该着手写本书第二版的时候，我反复问自己，为什么要继续写呢？最终，决定动笔最重要的因素是我从2008年第一版出版后陆续收到的数以百计的正面留言。我发现，出版一本书时得到大家的肯定就是一种回报！就冲这一点，最应该感谢的是那些读完第一版后给了我反馈的读者。没有你们给我动力，第二版也不会存在了。

当然写一本书只有动力还不行。之前提过，写书牵扯大量的时间。在这方面我是幸运的，我所在的Trivadis公司（我从1999年入职该公司）给了我全力支持，Trivadis不仅让我自身的技术得到提升，同时还鼓励我追求那些并非总有绝对把握的事情（比如写书）。所以第二个感谢应该送给Trivadis公司。

当你集中精力写一段文字超过一定的时间，有时会忽视一些显而易见的事情。基于这一点，我得说身边有几个人时常帮你检查你的工作是非常重要的。谨在此向技术评审人Alberto、Franco和Jože致以诚挚的谢意，是你们帮我极大地改进了本书的质量。当然，若有其他不足和错误都是我自己的责任。除了几位"官方的"技术评审人以外，还要感谢Dani Schnider、Franck Pachot、Randolf Geist和Tony Hasler等人，他们在阅读本书某些部分后给出了宝贵的评论和见解。

还要感谢Apress的工作人员在本书创作过程中给予的支持。尤其感谢Jonathan Gennick，他坚持认为创作第二版是明智的选择。

和第一版一样，本书出版的另一个核心人物是Curtis Gautschi。实际上，他再一次协助我校对了全书，尽管他并不能完全理解他读到的内容（据他声称）。非常感谢你，Curtis，这么多年来一直帮助我。

最后，特别感谢Jonathan和Cary为表示支持而为本书作序。你们在我职业生涯起步时激励了我，如今希望本书可以激励更多Oracle社区中的人做出正确的事情。

第一版致谢

许多人协助我写出了你手中的这本书。我由衷地感激他们。没有他们的帮助，这部作品就不会有机会面世。请允许我在跟各位分享这本书的简史时，感谢成就这一切的人们。

虽然当时我并没有意识到，但此书的写作与出版历程始于2004年7月16日，当时我正在为一个叫作“Oracle优化解决方案”的研讨会召开启动会议，与几个Trivadis的同事计划写一些材料。在会上，我们讨论了研讨会的目标和结构。那天以及随后几个月写下的研讨材料中产生的想法都用在了本书中。非常感谢当时与Arturo Guadagnin、Dominique Duay和Peter Welker的合作。我们当时一起写下的研讨材料，相信以今天的眼光来看也是一流的。除了他们几个，我还要感谢Guido Schmutz，他虽然只参加了启动会议，却强烈影响了我们处理研讨会中涉及的主题的方式方法。

2006年春天，也就是两年以后，我开始认真考虑要写这本书。我当时决定联系在Apress工作的Jonathan Gennick，告诉他我的想法并征询他的意见。从一开始，他就对我的提议很感兴趣，所以仅仅几个月后，我就决定将来在Apress出版此书。谢谢你，Jonathan，从一开始就支持我。此外，感谢所有为此书成功付梓而付出心血的Apress员工。我个人有幸与Sofia Marchant、Kim Wimpsett和Laura Esterman合作，但我知道还有其他很多人也同样做出了贡献。

有了想法和出版商并不足以写出一本书，你还需要时间，大量的时间。幸运的是，我的公司Trivadis支持并允许我花费时间在此书的创作上。在这里尤其要感谢Urban Lankes和Valentin De Martin。

当你写作时周围有人帮你仔细检查写下的内容也是至关重要的。非常感谢Alberto Dell’Era、Francesco Renne、Jože Senegacnik和Urs Meier这几位技术评审人，他们为帮助此书提高质量做出颇多贡献。当然，如有其他遗留问题都是我的责任。除技术评审外，我还要感谢Daniel Rey、Peter Welker、Philipp vondem Bussche-Hünnefeld及Rainer Hartwig，他们在阅读了本书部分内容后给出了宝贵的评论和见解。

此书出版的另一个核心人物是Curtis Gautschi。多年来，都是他帮忙校对并提高了我糟糕的英语。太感谢你了，Curtis，帮助了我这么多年。我承认，某一天我真得加强一下英语技能了。不过，我发现改进基于Oracle应用程序的性能比学外语更有意思（也更容易）。

在这里特别感谢Cary Millsap和Jonathan Lewis为本书作序。我知道这占去了你们很多宝贵的时间，非常感激二位。

同时特别感谢Grady Booch允许我在第1章中使用他的漫画。

最后，我要感谢这些年我有幸当过顾问的公司，感谢所有那些参加了我的课程和研讨会并提出很多好问题的人，感谢那些分享知识的Trivadis顾问。我从你们所有人当中获益良多。

引　言

Oracle数据库已经成长为超大型软件。这不仅意味着仅凭一己之力不再能够精通新版本提供的所有特性，同时也表明有一些特性很少会用到。实际上，在大多数情况下，能够掌握并利用其中一部分核心特性就足以高效、成功地使用Oracle数据库。所以在本书中，我根据经验，仅挑选出那些在诊断数据库相关性能问题时必然要用到的特性。

组织结构

本书分为四个部分。

第一部分涵盖了阅读本书剩余部分所需的基础知识。第1章不仅解释了为什么一定要在正确的时间使用正确的方法处理性能问题，还说明了为什么一定要了解业务需求和问题所在。这一章也描述了由数据库相关设计问题引发的一些常见的性能不佳的情况。第2章描述了数据库引擎在解析和执行SQL语句时所执行的操作，以及如何检测应用程序代码和数据库调用。另外，这一章也介绍了本书中常用的一些重要术语。

第二部分解释了如何在使用Oracle数据库的环境中处理性能问题。第3章描述如何借助SQL跟踪和PL/SQL探查器识别性能问题。第4章描述如何利用动态性能视图提供的信息，同时还将介绍几个经常与动态性能视图一起使用的工具和技术。第5章描述如何借助自动工作负载存储库AWR和Statspack来分析之前发生的性能问题。

第三部分描述负责将SQL语句生成执行计划的组件：查询优化器。第6章概述了查询优化器的功能及其实现方式。第7章和第8章描述什么是系统统计信息和对象统计信息，如何收集统计信息，以及统计信息对于查询优化器的重要性。第9章讲述如何通过配置路线图为查询优化器制定合理的配置。第10章描述获得、解释执行计划和评估执行计划效率所需了解的细节知识。

第四部分展示了Oracle数据库为高效执行SQL语句提供的特性。第11章描述了如何通过Oracle数据库提供的相关技术去影响查询优化器生成执行计划。第12章描述了如何识别、解决以及排除由解析引发的性能问题。第13章描述访问数据的多种方法以及如何在其中选择合适的。第14章讨论如何高效联接多个数据集。第15章描述类似并行处理、物化视图和结果集缓存这样的高级调优技术。第16章解释为什么优化数据库的物理设计如此重要。

目标读者

本书的目标读者是那些因在应用程序中使用了Oracle数据库而涉及诊断性能问题的性能专家、应

用程序开发人员和数据库管理员。

本书不需要某些具体的优化方面的知识。但是，希望读者具有Oracle数据库相关的应用知识并熟练掌握SQL。本书某些章节会涉及关于具体的编程语言（如PL/SQL、Java、C#、PHP以及C等）的一些特性。之所以提及这些特性，仅是为了照顾不同的应用开发人员在使用不同的编程语言时所体现的信息差异，你可以挑选自己正在使用的或者感兴趣的语言。

涵盖哪些版本

本书涉及的大部分重要概念都不依赖于你所使用的Oracle数据库版本。然而不可避免地，当讨论具体的实现细节时，某些内容是与版本相关的。本书主要讨论的是目前可用的版本，包括从Oracle Database 10*g* R2至Oracle Database 12*c* R1，如下所示。

- Oracle Database 10*g* R2，包含的版本至10.2.0.5.0
- Oracle Database 11*g* R1，包含的版本至11.1.0.7.0
- Oracle Database 11*g* R2，包含的版本至11.2.0.4.0
- Oracle Database 12*c* R1，版本 12.1.0.1.0

注意，粒度是补丁集（patch set）级别，因此，本书不讨论安全补丁和捆绑补丁（bundle patch[①]）所带来的变化。如果没有明确说明某一特性仅适合某一特定版本，那么它对所有提到的版本都有效。

在线资源

可以在网站http://top.antognini.ch上下载本书引用的文件，也可以在其中找到勘误和补充资料。另外，如果有关于本书的任何类型的反馈意见或问题，请发送到top@antognini.ch。

与第一版的不同之处

本书修订的主要目标包括以下各项。

- 增加关于Oracle Database 11*g* R2和Oracle Database 12*c* R1的内容。
- 删掉关于Oracle Database 9*i*和Oracle Database 10*g* R1的内容。
- 补上第一版遗漏的内容，例如层次剖析工具、活动会话历史（ASH）、AWR及Statspack等。
- 当涉及具体的编程语言特性时加入一些有关PHP的知识。
- 为提高可读性重新组织了部分素材，例如，将系统和对象统计信息拆分为两章。
- 修复勘误，改进行文组织。

① 一种临时补丁，包含许多重要的bug修复，但是没有PSU多，主要供Windows平台使用。这里指未考虑小版本号差异，如10.2.0.5.6或10.2.0.5.12。——译者注

目录

第一部分 基础

第 1 章 性能问题……2
1.1 需要为性能做规划吗……2
1.1.1 需求分析……2
1.1.2 分析与设计……4
1.1.3 编码和单元测试……4
1.1.4 集成和验收测试……6
1.2 为性能而设计……6
1.2.1 缺乏数据库逻辑设计……6
1.2.2 实现通用表……7
1.2.3 未使用约束加强数据完整性……7
1.2.4 缺乏数据库物理设计……7
1.2.5 未正确选择数据类型……8
1.2.6 未正确使用绑定变量……8
1.2.7 未利用数据库高级特性……8
1.2.8 未使用 PL/SQL 进行以数据为中心的处理……9
1.2.9 执行不必要的提交……9
1.2.10 持续打开和关闭数据库连接……9
1.3 你真的面临性能问题吗……9
1.3.1 系统监控……10
1.3.2 响应时间监控……10
1.3.3 强迫性调优障碍……10
1.4 如何处理性能问题……11
1.4.1 业务视角和系统视角……11
1.4.2 问题的编录……12
1.4.3 解决问题……12
1.5 小结……15
第 2 章 关键概念……16
2.1 选择率和基数……16
2.2 什么是游标……17
2.3 游标的生命周期……18
2.4 解析的工作原理……20
2.4.1 可共享游标……22
2.4.2 绑定变量……25
2.5 读写数据块……35
2.6 检测……36
2.6.1 应用程序代码……37
2.6.2 数据库调用……39
2.7 小结……42

第二部分 识别

第 3 章 分析可重现的问题……45
3.1 跟踪数据库调用……45
3.1.1 SQL 跟踪……45
3.1.2 跟踪文件的结构……57
3.1.3 使用 TRCSESS……59
3.1.4 探查器……60
3.1.5 使用 TKPROF……60
3.1.6 使用 TVD$XTAT……70
3.2 探查 PL/SQL 代码……79
3.2.1 使用 DMBS_HPROF……79
3.2.2 使用 DBMS_PROFILER……85
3.2.3 触发探查器……89
3.3 小结……90
第 4 章 实时分析不可重现的问题……91
4.1 分析路线图……91
4.2 动态性能视图……93
4.2.1 操作系统统计信息……93

4.2.2 时间模型统计信息……94
4.2.3 等待级别和等待事件……96
4.2.4 系统和会话统计信息……100
4.2.5 度量值……101
4.2.6 当前会话状态……102
4.2.7 活动会话历史……103
4.2.8 SQL 语句统计信息……111
4.2.9 实时监控……112
4.3 使用 Diagnostics Pack 和 Tuning Pack 进行分析……115
4.3.1 数据库服务器负载……115
4.3.2 系统级别分析……116
4.3.3 会话级别分析……120
4.3.4 SQL 语句信息……122
4.4 不使用 Diagnostics Pack 进行分析……125
4.4.1 数据库服务器负载……125
4.4.2 系统级别分析……126
4.4.3 会话级别分析……129
4.4.4 SQL 语句信息……130
4.5 小结……131
第 5 章 不可重现问题的事后分析……132
5.1 知识库……132
5.2 自动工作负载存储库……133
5.2.1 执行配置……133
5.2.2 捕获快照……134
5.2.3 管理基线……135
5.3 Statspack……136
5.3.1 执行安装……137
5.3.2 配置存储库……137
5.3.3 捕获和清除快照……138
5.3.4 管理基线……139
5.4 使用 Diagnostics Pack 进行分析……140
5.5 不使用 Diagnostics Pack 进行分析……140
5.6 小结……145
第三部分 查询优化器
第 6 章 查询优化器简介……148
6.1 基础知识……148
6.2 体系结构……150
6.3 查询转换……152
6.3.1 计数转换……152
6.3.2 公共子表达式消除……153
6.3.3 “或”扩张……153
6.3.4 视图合并……154
6.3.5 选择列表裁剪……155
6.3.6 谓词下推……156
6.3.7 谓词迁移……158
6.3.8 非重复放置……158
6.3.9 非重复消除……159
6.3.10 Group-by 放置……159
6.3.11 Order-By 消除……160
6.3.12 子查询展开……160
6.3.13 子查询合并……161
6.3.14 使用窗口函数移除子查询……162
6.3.15 联接消除……162
6.3.16 联接因式分解……163
6.3.17 外联接转内联接……163
6.3.18 完全外联接……164
6.3.19 表扩张……164
6.3.20 集合操作联接转变……165
6.3.21 星型转换……166
6.3.22 物化视图查询重写……166
6.4 小结……166
第 7 章 系统统计信息……167
7.1 dbms_stats 包……167
7.2 有哪些系统统计信息可用……168
7.3 收集系统统计信息……170
7.3.1 无工作负载统计信息……170
7.3.2 工作负载统计信息……171
7.3.3 在无工作负载统计信息和工作负载统计信息之间进行选择……174
7.4 还原系统统计信息……174
7.5 使用备份表……175
7.6 管理操作的日志记录……176
7.7 对查询优化器的影响……177
7.8 小结……182

第 8 章　对象统计信息 183
8.1　dbms_stats 包 183
8.2　有哪些对象统计信息可用 185
8.2.1　表统计信息 186
8.2.2　列统计信息 187
8.2.3　直方图 189
8.2.4　扩展统计信息 200
8.2.5　索引统计信息 205
8.2.6　分区对象统计信息 206
8.3　收集对象统计信息 207
8.3.1　目标对象 208
8.3.2　收集选项 212
8.3.3　备份表 217
8.4　配置 dbms_stats 包 218
8.4.1　传统方式 218
8.4.2　现代方式 219
8.5　处理全局临时表 221
8.6　处理挂起的对象统计信息 222
8.7　处理分区对象 223
8.7.1　挑战 223
8.7.2　增量统计信息 226
8.7.3　复制统计信息 228
8.8　调度对象统计信息的收集 229
8.8.1　10*g* 方式 229
8.8.2　11*g* 和 12*c* 方式 231
8.9　还原对象统计信息 232
8.10　锁定对象统计信息 234
8.11　比较对象统计信息 236
8.12　删除对象统计信息 238
8.13　导出、导入、获取和设置对象统计信息 239
8.14　管理操作的日志记录 239
8.15　保持对象统计信息为最新的策略 241
8.16　小结 242
第 9 章　配置查询优化器 243
9.1　配置还是不配置 243
9.2　配置路线图 244
9.3　设置正确的参数 245
9.3.1　查询优化器参数 246
9.3.2　PGA 管理 260
9.4　小结 266
第 10 章　执行计划 267
10.1　获取执行计划 267
10.1.1　EXPLAIN PLAN 语句 267
10.1.2　动态性能视图 270
10.1.3　自动工作负载存储库和 Statspack 272
10.1.4　跟踪工具 274
10.2　dbms_xplan 包 277
10.2.1　输出 277
10.2.2　display 函数 281
10.2.3　display_cursor 函数 286
10.2.4　display_awr 函数 288
10.3　解释执行计划 289
10.3.1　父-子关系 290
10.3.2　操作的类型 292
10.3.3　独立操作 292
10.3.4　迭代操作 295
10.3.5　无关联组合操作 295
10.3.6　关联组合操作 297
10.3.7　分而治之 305
10.3.8　特殊情况 307
10.3.9　自适应执行计划 310
10.4　识别低效的执行计划 314
10.4.1　错误的估算 314
10.4.2　未识别限制条件 316
10.5　小结 317

第四部分　优　化

第 11 章　SQL 优化技巧 320
11.1　修改访问结构 321
11.1.1　工作原理 321
11.1.2　何时使用 322
11.1.3　陷阱和谬误 322
11.2　修改 SQL 语句 322
11.2.1　工作原理 322
11.2.2　何时使用 323

11.2.3 陷阱和谬误 ······324
11.3 hint ······324
11.3.1 工作原理 ······324
11.3.2 何时使用 ······330
11.3.3 陷阱和谬误 ······330
11.4 修改执行环境 ······332
11.4.1 工作原理 ······332
11.4.2 何时使用 ······334
11.4.3 陷阱和谬误 ······334
11.5 存储概要 ······334
11.5.1 工作原理 ······335
11.5.2 何时使用 ······343
11.5.3 陷阱和谬误 ······343
11.6 SQL 配置文件 ······344
11.6.1 工作原理 ······345
11.6.2 何时使用 ······357
11.6.3 陷阱和谬误 ······357
11.7 SQL 计划管理 ······358
11.7.1 工作原理 ······359
11.7.2 何时使用 ······372
11.7.3 陷阱和谬误 ······372
11.8 小结 ······373

第 12 章 解析 ······374

12.1 识别解析问题 ······374
12.1.1 快速解析 ······375
12.1.2 长解析 ······380
12.2 解决解析问题 ······381
12.2.1 快速解析 ······381
12.2.2 长解析 ······387
12.3 避开解析问题 ······387
12.3.1 游标共享 ······388
12.3.2 服务器端语句缓存 ······390
12.4 使用应用编程接口 ······392
12.4.1 PL/SQL ······392
12.4.2 OCI ······395
12.4.3 JDBC ······396
12.4.4 ODP.NET ······398
12.4.5 PHP ······399
12.5 小结 ······400

第 13 章 优化数据访问 ······401

13.1 识别次优访问路径 ······401
13.1.1 识别 ······401
13.1.2 误区 ······403
13.1.3 原因 ······405
13.1.4 解决方案 ······406
13.2 弱选择性的 SQL 语句 ······409
13.2.1 全表扫描 ······409
13.2.2 全分区扫描 ······411
13.2.3 范围分区 ······411
13.2.4 散列和列表分区 ······422
13.2.5 复合分区 ······422
13.2.6 设计要素 ······424
13.2.7 全索引扫描 ······426
13.3 强选择性的 SQL 语句 ······429
13.3.1 Rowid 访问 ······429
13.3.2 索引访问 ······430
13.3.3 单表散列群集访问 ······468
13.4 小结 ······470

第 14 章 优化联接 ······471

14.1 定义 ······471
14.1.1 联接树 ······471
14.1.2 联接的类型 ······475
14.1.3 限制条件与联接条件 ······478
14.2 嵌套循环联接 ······479
14.2.1 概念 ······479
14.2.2 两表联接 ······480
14.2.3 四表联接 ······481
14.2.4 缓冲区缓存预取 ······482
14.3 合并联接 ······484
14.3.1 概念 ······484
14.3.2 两表联接 ······485
14.3.3 四表联接 ······488
14.3.4 工作区 ······489
14.4 散列联接 ······494
14.4.1 概念 ······494
14.4.2 两表联接 ······495
14.4.3 四表联接 ······496
14.4.4 工作区 ······498

14.4.5　索引联接 ……498
14.5　外联接……499
14.6　选择联接方法 ……499
14.6.1　First-Rows 优化……500
14.6.2　All-Rows 优化 ……500
14.6.3　支持的联接方法……500
14.6.4　并行联接 ……500
14.7　分区智能联接……501
14.7.1　完全智能化分区连接……501
14.7.2　部分智能化分区联接……504
14.8　星型转换……505
14.9　小结 ……511
第 15 章　数据访问和联接优化之外……512
15.1　物化视图……512
15.1.1　工作原理 ……512
15.1.2　何时使用 ……530
15.1.3　陷阱和谬误 ……531
15.2　结果缓存……531
15.2.1　工作原理 ……532
15.2.2　何时使用 ……538
15.2.3　陷阱和谬误 ……538
15.3　并行处理……539
15.3.1　工作原理 ……540
15.3.2　何时使用 ……567
15.3.3　陷阱和谬误 ……567
15.4　直接路径插入 ……571
15.4.1　工作原理 ……572
15.4.2　何时使用 ……574
15.4.3　陷阱和谬误 ……574
15.5　行预取 ……575
15.5.1　工作原理 ……575
15.5.2　何时使用 ……579
15.5.3　陷阱和谬误 ……580
15.6　数组接口……580
15.6.1　工作原理 ……580
15.6.2　何时使用 ……583
15.6.3　陷阱和谬误 ……583
15.7　小结 ……583
第 16 章　优化物理设计……584
16.1　最优列顺序……584
16.2　最优数据类型……586
16.2.1　数据类型选择中的陷阱……586
16.2.2　数据类型选择最佳实践……589
16.3　行迁移和行链接……591
16.3.1　迁移与链接 ……591
16.3.2　问题描述 ……593
16.3.3　问题识别 ……593
16.3.4　解决方案 ……594
16.4　块争用 ……594
16.4.1　问题描述 ……594
16.4.2　问题识别 ……595
16.4.3　解决方案 ……599
16.5　数据压缩……602
16.5.1　概念……602
16.5.2　要求……603
16.5.3　方法……603
参考文献……606

Part 1 第一部分

基础

Chi non fa e'fondamenti prima, gli potrebbe con una grande virtú farli poi, ancora che si faccino con disagio dello architettore e periculo dello edifizio.

一个人如果没有先打好基础，事后也还有可能运用其卓越的能力进行巩固，但这对于建筑师来说很困难，对于建筑物来说也很危险。[①]

——尼科洛·马基雅维利，《君主论》，1532 年

① 英文版由 W. K. Marriott 翻译，链接地址：http://www.gutenberg.org/files/1232/1232-h/1232-h.htm。

第 1 章 性能问题

太多时候，优化工作在应用程序开发结束以后才开始。这种做法是不可取的，因为它会导致人们以为性能并不像应用程序的其他关键需求那样重要。性能并不只是应用程序的一种可选指标，而是一个关键指标。糟糕的性能不仅有损应用程序的可接受程度，而且通常还会降低用户的工作效率，导致投资回报率较低。IBM在20世纪80年代早期所做的多项研究表明，应用程序的性能与用户的工作效率有着密切的关系：系统处理事务的效率越低，用户的思考时间就会越长，发生错误的概率就会越高，这是用户长时间等待之后注意力下降的必然结果。此外，性能糟糕的应用程序还往往导致更高的软件成本、硬件成本及维护成本。基于上述原因，本章将主要讨论为何性能规划如此重要，哪些是最常见的导致性能欠佳的设计失误，以及如何知道一个应用程序出现了性能问题。而后，本章将讨论出现性能问题时该如何进行处理。

1.1 需要为性能做规划吗

在软件工程领域，用于管理项目开发的模型各式各样。不管是类似瀑布模型的顺序型生命周期，还是类似敏捷开发的迭代型生命周期，都需要经历几个共同阶段（见图1-1）。在项目开发过程中，这些阶段可能只出现一次（如瀑布模型），也可能出现多次（如迭代模型）。



图1-1　应用程序开发的主要阶段

如果仔细分析以上每个阶段所需开展的工作，你也许会注意到每个阶段都有性能要求。即便如此，在实际开发过程中，开发团队还是会时常忘记性能要求，直到性能问题浮现出来。而那时也许为时已晚。因此，本章接下来的部分将从性能的角度出发，介绍在下一次开发应用程序时不应该再忽视的内容。

1.1.1 需求分析

简单来讲，*需求分析*（requirements analysis）就是确定应用程序的主要目标以及藉此期望达成的目的。进行需求分析之前，通常要对多个利益相关方进行调研。这一步十分必要，因为单独一方不太可能确定所有的业务需求和技术需求。由于需求的来源不一，因此必须对需求进行仔细分析，尤其需

要找出不同需求间是否存在潜在冲突。在进行需求分析时，不仅要关注应用程序需要提供的功能，仔细确定这些功能的使用率也是至关重要的。对于每一个具体的功能，需要预估与之交互的用户[①]数量、用户的使用频率以及每次使用时的预期响应时间。换句话说，你必须确定预期的性能指标。

响应时间

从请求进入系统或者功能单元到其离开的时间间隔叫作**响应时间**（response time）。响应时间可以进一步分解为**服务时间**（service time）（系统处理请求所需时间）和**等待时间**（wait time）（请求等待处理的时间）。等待时间在排队论中又称为**排队延迟**（queueing delay）。

响应时间 = 服务时间 + 等待时间

如果考虑到用户在执行动作（例如单击某个按钮）时某个请求进入系统，而用户在收到系统对于这个动作所做出的相应反应时该请求离开系统，之间的这段时间间隔可以叫作**用户响应时间**。换句话讲，用户响应时间是指从用户角度来计算的处理一个请求所需的时间。

有些情况下，如Web应用，一般不考虑用户响应时间，因为在请求抵达应用程序的第一个组件（通常是网络服务器）之前一般无法对它们进行跟踪。此外，大多数情况下保证用户响应时间是不可能的，因为应用程序提供商并不负责用户程序（通常是浏览器）与系统程序第一个组件之间的网络。此时，测量并保证从请求进入系统的第一个组件到离开的时间间隔更为合理。这一时间间隔称为**系统响应时间**。

表1-1是由JPetStore[②]提供的一些操作的预期性能数据样例。对于每项操作，该表给出了系统对于收到的90%和99.99%的请求所能保证的系统响应时间。多数情况下，要保证系统对于所有请求（即100%的请求）的性能，要么不可能，要么需要高额投入。所以，最常见的做法是指明一小部分请求可能无法达到所需的响应时间。由于系统负载随日常运行发生变化，因此该表用两个值来表示最大到达率。本例中，最高的事务率预计出现在白天。但在其他情况下，例如将批量作业放在夜间进行时，可能会有所不同。

表1-1 由某网络商店提供的典型操作的性能数据

操作类型	最长响应时间（秒）		最大到达率（事务数/分钟）	
	90%	99.99%	0~7	8~23
注册/修改个人资料	2	5	1	2
登录/退出	0.5	1	5	20
检索商品	1	2	60	240
显示商品概览	1	2	30	120
显示商品细节	1.5	3	10	36
从购物车添加/更改/移除商品	1	2	4	12

① 注意：这里的“用户”并不总是指人。举个例子，如果你正在定义一个网络服务需求，很可能它的“用户”只是其他应用程序。

② JPetStore是由Spring Framework等提供的一个示例应用程序。登录可下载或获取更多信息。

（续）

操作类型	最长响应时间（秒）		最大到达率（事务数/分钟）	
	90%	99.99%	0~7	8~23
显示购物车	1	3	8	32
提交/确认订单	1	2	2	8
显示订单	2	5	4	16

这些性能需求不仅仅作为核心要素贯穿于应用程序开发的各个阶段（见后面几节），稍后也可将其作为定义服务级别协议以及制定容量规划的基础。

服务级别协议

服务级别协议（SLA）是用来明确服务提供商和用户之间关系的契约。它描述的内容包括服务项目，其在运行时间和停机时间的可用性、响应时间、客户支持水平，以及一旦服务提供商无法履行协议时相应的处理方式。

只有在能够验证响应时间的情况下，才能根据响应时间制定服务级别协议。这需要定义清晰的、可测量的性能数据以及与之相关的目标。这些性能数据通常被称作**关键性能指标**（Key Performance Indicator，KPI）。最理想的情况是使用一种监控工具收集、存储和评估这些指标数据。事实上，这样做的目的不仅是为了在某个目标没有达到时进行标识，还能为日后出具报告和制定容量规划而记录下依据。为了收集这些性能数据，可以采用两种主要的技术手段。第一种是利用监测代码（instrumentation code）的输出结果（详见第2章）；第二种是使用响应时间监控工具（参见1.3.2节）。

1.1.2　分析与设计

架构设计师根据需求设计解决方案。开始的时候，为了定义架构，需要考虑所有的需求。事实上，对于一个需要承受高负载的应用程序，在设计之初就应考虑负载需求。当设计时用到诸如并行化、分布式计算或结果重用等技术时更应如此。例如，设计一个支持少数用户每分钟执行十几个事务的C/S应用程序，与设计一个支持成千上万用户每秒执行数百个事务的分布式系统完全是两码事。

有时需求也会通过在某一资源的使用上施加限制来影响架构。例如，如果一个应用程序用于通过低速网络连接到服务器的移动设备，那么其架构设计必须考虑能够支持较大延迟和较低吞吐量。通常，架构设计师不仅需要预见到一个方案可能出现的瓶颈，还要衡量这些瓶颈是否会危及需求的实现。如果架构设计师没有掌握足够的信息来进行这样的关键预先评估，就应该开发一个或甚至多个原型。在这方面，如果没有前一阶段收集的性能数据，将很难做出明智的决定。我所说的明智的决定是指那些能够实现以最小投资支撑预期负载的架构/设计的决定：简单方案处理简单问题，精简方案处理复杂问题。

1.1.3　编码和单元测试

专业开发人员编写的代码应具有下面这些特点。

- **稳定性**：拥有应对意外情况的能力是所有软件都应具备的特性。为了达到预期质量，必须定期进行单元测试。这一点对于迭代型生命周期尤为重要。实际上，在这类模型中，快速重构既有

代码的能力是不可或缺的。例如，在调用某个子程序时，如果传递的参数值超过指定范围，系统就必须能够做出相应处理而不至于崩溃。如有必要，应该同时生成有意义的错误信息。

- **可维护性**：能够长期运行、结构良好、已文档化的可读代码比没有文档化的糟糕代码维护起来要容易得多（维护费用也更低）。例如，有人将多个操作写成单独一行晦涩的代码，这样的开发人员其实选错了展示才华的方式。
- **运行速度**：代码应该进行优化，以期尽可能提高运行速度。在预期负载很高的情况下更应如此。代码应该具有可伸缩性，进而能够利用额外的硬件资源应对用户或事务的不断增加。例如，应该避免不必要的操作、串行程序，以及低效或不适合的算法。然而，一定不要掉进过早优化的陷阱。
- **精明的资源利用**：代码应尽最大可能利用可访问资源。注意，这并不总是意味着使用最少的资源。比如，应用程序使用并行化操作比串行化操作要消耗更多的资源，但是有时候并行化也许是解决苛刻负载的唯一途径。
- **安全性**：毋庸置疑，代码要拥有保证数据机密性和完整性，以及对用户进行验证和授权的能力。有时，不可抵赖性也是需要考虑的问题。例如，可能需要用到数字签名来防止终端用户否认通信或合同的有效性。
- **可检测性**：检测的目的有两方面。其一，更易于分析出现的功能问题和性能问题（即使是精心设计的系统也无法避免这些问题的出现）；其二，有策略地添加代码以提供应用程序的性能信息。例如，通常情况下，添加用以获取某一操作所耗时间的代码非常简单。这是一个验证应用程序是否能够满足必要性能需求的简单有效的办法。

不仅这些特性彼此之间确实存在一些冲突，而且预算通常是有限的（有时甚至非常有限）。因此，我们通常有必要在这些特性之间做个优先级排序，在其中找到平衡点，以便在有限的预算下实现预期的需求。

过早优化

过早优化是一个有争议性的话题，这（或许）源于Donald Knuth的名言“过早优化是万恶之源”。基于这句话的一个误解是，认为程序员在编码时应该完全忽略优化的事情。在我看来，这种想法是错误的。为了避免断章取义，让我们来看看这句名言的前因后果：

> “对于‘效率’一词的过度追求无疑会导致滥用。程序员们浪费大量的时间思考或者担心他们程序中的非关键部分的运行速度，这些追求效率的做法其实会对调试和维护造成极大的负面影响。我们应该忽略微小的效率因素，在约97%的时间中，过早优化是万恶之源。但是我们不应该错过那关键的3%。一个优秀的程序员不会因为这样的理由而自我满足。他知道要仔细检查核心代码，但前提是那些核心代码是经过认可的。预先判断程序的核心部分往往是一个错误，因为很多程序员的一个共同经历是：在使用测量工具后，他们发现靠直觉的猜测是错误的。”

我对Knuth的文章的理解是：程序员在编码时，不应该去关注那些只能产生局部影响的细微优化。相反，他们应该关心影响全局的优化手段，比如系统设计、实现所需功能所使用的算法，或者在哪个层面（SQL、PL/SQL、程序语言）的哪些指标需要进行特殊处理。局部优化应该等到测量工具显示某一模块的执行时间过长之后再进行。因为优化是局部的，所以并不影响系统的总体设计。

1.1.4 集成和验收测试

集成和验收测试的目的是验证应用程序的功能需求、性能需求以及系统稳定性。性能测试和功能测试同等重要，这一点无论如何强调都不过分。从各方面来看，一个性能差的应用程序和没有实现功能需求的应用程序一样糟糕。在这两种情况下，应用程序都是无用的。然而，只有明确定义过性能需求才有可能去验证它。

缺少正式的性能需求会导致两个主要问题。第一，极有可能在集成和验收测试阶段没有执行严格的、有条不紊的压力测试，这样，应用程序就会在不知道是否能够支持预期负载的情况下交付生产；第二，就性能而言，无法明确什么样的表现可以接受，什么样的表现不能接受。通常，只有在极端情况下（也就是说，性能非常好或非常糟糕），不同的人才会达成统一意见。如果无法达成共识，冗长、恼人、徒劳的会议以及人际冲突就会随之出现。

在实践中，设计、实现和执行良好的集成和验收测试来验证应用程序的性能表现并非易事。要想取得成功必须面对下面三个主要挑战。

- 设计压力测试时应该考虑能够产生典型的负载。对此主要有两种方法：一是让真实的用户做真实的工作；二是用工具来模拟用户。两种方法各有优缺点，应该根据具体情况进行具体分析。某些情况下，两种方法可同时用于对不同模块进行压力测试，或者用两种方法进行互补。
- 要产生典型的负载，就需要典型的测试数据。不仅数据行的数量和大小要符合预期的量，数据分布情况和数据内容也应与真实数据一致。例如，如果属性中含有城市名称，那么用真实的城市名称就比用像Aaaacccc或Abcdefghij这样的字符串要好得多。这样做很重要，因为很多情况下应用程序和数据库都会因为不同的数据导致不同的表现（例如，索引或作用于数据的函数）。
- 测试的基础设施要尽可能与生产环境的基础设施保持一致。这对于高度分布的系统和需要与许多其他系统协作的系统来说尤其困难。

在顺序型生命周期模型中，项目开发接近尾声时才进入集成和验收测试阶段。如果导致性能问题的重大系统架构缺陷此时才被发现，问题会比较棘手。为避免这样的问题，在编码和单元测试阶段也应该进行压力测试。注意，迭代型生命周期模型不存在这种问题。事实上，根据“迭代型生命周期模型”的定义，每一次迭代都应该执行压力测试。

1.2 为性能而设计

考虑到应用程序应该围绕性能而设计，详细介绍一种设计方法会非常有用。但是本书的焦点是问题诊断。为此，这里只粗略介绍容易导致性能欠佳的十个最常见的与数据库相关的设计问题。

1.2.1 缺乏数据库逻辑设计

曾经，大家认为每个开发项目理所当然都需要数据架构师的参与。通常这个人不仅负责数据和数据库设计，也是负责应用程序整体架构和设计的团队的一员。这个人通常拥有极为丰富的数据库相关经验。他十分清楚如何进行数据库设计以保证数据的完整性和性能。

遗憾的是，现在大家并不总是这样认为。我常常发现很多项目根本没有经过正规的数据库设计。

客户端或中间层设计由应用程序开发人员完成，他们有时两者兼顾。然后，突然间数据库设计就由像持久层框架这样的工具生成了。在这样的项目中，数据库被视为存储数据的“哑设备”。这种对于数据库的理解是错误的。

1.2.2 实现通用表

每一位CIO都梦想着应用程序可以轻松应对新出现的或变化的需求。灵活性是这里的关键词。这些梦寐以求的应用程序有时可以通过使用通用数据库设计来实现。这种情况下，添加新数据只需更改配置而无需更改数据库对象本身。

有两种主要的数据库设计可用于实现这样的灵活性。

- **实体-属性-值（EAV）模型**：就像命名中暗示的那样，描述每个信息时都至少使用三列：实体、属性和值。每个组合定义一个与某一实体关联的具体属性的值。
- **基于XML的设计**：每张表只包含为数不多的几列。其中，必会出现这样两列：一个标识符，以及一个用来存储几乎其他所有值的XML列。有时也会用到其他列来存储元数据（例如，谁在何时做了最后一次修改）。

从性能的角度看，这样设计的问题在于它们（至少）不是最优化的。事实上，灵活性和性能息息相关却又相互矛盾。某些情况下即使未达到最优的性能可能也足够了，但是在其他情况下则可能是灾难性的。因此，应该仅在确保可以达到所需性能时才进行灵活设计。

1.2.3 未使用约束加强数据完整性

约束（主键、唯一键、外键、非空约束和检查约束）不仅是保证数据完整性的基础，而且也广泛用于查询优化器生成执行计划的过程中。如果不使用约束，查询优化器就无法利用很多优化技术。除此之外，在应用程序级别上检查约束将导致更多的代码编写和测试工作，并为数据完整性带来潜在问题，因为总是可以在数据库层面人为修改数据。而且，在应用程序级别上检查约束通常需要消耗更多的资源，并导致更少的可扩展的加锁方案（比如锁住整张表，而不是让数据库只对某几行进行加锁）。为此，我强烈建议应用程序开发人员在数据库级别定义所有已知的约束。

1.2.4 缺乏数据库物理设计

很多项目没有充分利用Oracle数据库提供的特性，直接由逻辑设计映射出物理设计，这种情形并不罕见。最显而易见的例子是每个关系都直接映射到一张堆表中。从性能的角度看，这并不是最佳方法。在很多情况下，索引组织表（IOT）、索引群集或散列群集可能会提供更好的性能。

Oracle数据库提供的索引类型远远不止常用的B树索引和位图索引。根据具体的情况，压缩索引、反转键索引、基于函数的索引、语义索引或文本索引可能对于改进性能非常有价值。

对于大型数据库，分区选项的实现至关重要。大多数的数据库管理员都能意识到这个选项及其作用。这里的一个常见问题是开发人员认为分区表不影响数据库物理设计。有时事实如此，有时则不然。因此我强烈建议在项目初期就开始计划如何使用分区。

在开发新应用程序期间，另一个需要经常应对的问题是定义和实现一个合理的数据归档思路。推迟这项工作通常是不允许的，因为它可能会影响数据库物理设计（如果不影响逻辑设计的话）。

1.2.5 未正确选择数据类型

近些年来，我目睹了数据库物理设计中一种令人不安的趋势。这种趋势可称为错误数据类型选择（像存储日期时使用VARCHAR2替代DATE或者TIMESTAMP类型）。乍看起来，选择数据类型似乎是非常简单的决定。然而，有许多目前正在运行的系统由于选择了错误的数据类型而备受折磨，这种系统的数量不容低估。

在数据类型选择错误方面主要存在以下四个问题。

- **数据验证的错误或缺乏：**数据库引擎必须有能力验证存储在数据库中的数据。例如，你应该避免使用字符串类型存储数字值。这样做会需要外部验证，引发类似于1.2.3节中描述的问题。
- **丢失信息：**在由原始（正确的）数据类型向（错误的）数据库数据类型转化期间，会发生信息丢失。例如，大家想象一下，用DATE数据类型替代TIMESTAMP WITH TIME ZONE数据类型去存储某个事件发生的日期和时间时会出现什么情况。小数位的秒和时区信息会发生丢失。
- **出现与预期不符的结果：**对数据的顺序有严格要求的操作和功能可能引发意想不到的结果，这是因为每种数据类型相关的具体对照语义是不同的。典型案例是范围分区表和ORDER BY子句相关的问题。
- **查询优化器异常：**查询优化器可能会因为错误的数据类型选择做出错误的估计，进而可能无法选择最优的执行计划。这不是查询优化器的过错。问题在于查询优化器并未获得全部所需信息，从而导致其无法正常工作。

总而言之，你有充分的理由对数据类型做出正确选择。这样做可能会帮你避免很多问题。

1.2.6 未正确使用绑定变量

从性能的角度看，绑定变量既有优势也有劣势。绑定变量的优势是它允许在库缓存中共享游标，从而避免了硬解析和相关开销；其劣势是在用于WHERE子句时（也仅在被用于WHERE子句时），有时会导致查询优化器无法获取关键信息。对于查询优化器来说，要为每个SQL语句都生成最优的执行计划，使用字面值替代绑定变量会好很多。第2章会详细讨论这个话题。

从安全的角度来看，绑定变量预防了与SQL注入攻击相关的风险。实际上，不可能通过给绑定变量传递一个值来更改SQL语句的语法。

1.2.7 未利用数据库高级特性

Oracle数据库是一款提供诸多高级特性的高端数据库引擎，可以极大地降低开发成本，更不用说在提高性能时节省的调试和bug修复的成本。应尽可能利用这些特性来提高投资回报率。尤其是要避免重新开发已经可用的特性（例如，不要创建自己的队列系统，因为数据库可以直接为你提供）。即便如此，第一次使用某个特性时也要特别当心，尤其在使用数据库版本的新增特性时更应如此。不仅应该仔细测试该特性是否满足需求，还要核实它的稳定性。

针对数据库高级特性最常见的争论是，当应用程序使用它们以后便与所用的数据库品牌联成一体，未来无法轻松转换成其他数据库。这是事实。但是，不管怎样，大多数公司极少会去更换某个应用程序的数据库引擎。相比只更换数据库引擎，他们更愿意更换整套应用程序。

我建议仅在有充分理由的情况下才去做独立于数据库的应用程序设计。如果真有某种原因必须做独立于数据库的设计，回过头去重新阅读1.2.2节中关于权衡灵活性与性能的讨论。那个讨论也适用于当前的情况。

1.2.8 未使用 PL/SQL 进行以数据为中心的处理

上一节讲到数据库高级特性的使用。有一种特别情况，就是使用PL/SQL实现大量数据的批处理。最常见的案例是抽取-转换-加载（ETL）过程。在这个过程中，当抽取和加载的对象是同一数据库时，从性能的角度来看，转换阶段不去使用管理来源和目标数据的数据库引擎所提供的SQL和PL/SQL简直愚蠢至极。遗憾的是，几个主流ETL工具的体系结构正好会导致这样的愚蠢行为。也就是说，将数据从数据库中抽取出来（而且经常被移至另一个服务器），进入转换阶段，然后将结果数据加载回原来的同一个数据库。基于这个原因，像Oracle这样的供应商开始提供在数据库内部执行转换的工具。为了与ETL工具有所区分，这种工具通常称为ELT。为了达到最佳性能，建议尽可能以数据为中心进行数据处理。

1.2.9 执行不必要的提交

提交是串行化的操作（原因很简单：只有一个LGWR进程负责将数据写入到重做日志文件）。不言而喻，每个串行化操作都影响可伸缩性。因此，串行化是不受欢迎的，应该尽可能最小化。一个办法是将几个无关事务放在一起执行。典型案例是批量作业加载很多行数据。与每次插入后提交相比，批量提交插入的数据要好得多。

1.2.10 持续打开和关闭数据库连接

打开一个数据库连接会在数据库服务器端相应地打开一个专有进程，这不是一个轻量级操作。不要低估这种做法所需的时间和资源。最坏的情形是，Web应用会为每个涉及数据库访问的请求都打开和关闭一个数据库连接。这样的方法是极其不适宜的。这种情形下使用连接池是重中之重。通过使用连接池，可以避免不断地启动和停止专有服务进程，从而避免涉及的所有开销。

1.3 你真的面临性能问题吗

早晚都得讨论应用的性能，这可能是一个很好的机会。如果你像前面章节提到的那样，仔细定义过性能需求，那么很容易就可以确定应用程序是否正在遭遇性能问题。如果没有仔细定义过性能需求，那么答案很大程度上会取决于回答这个问题的人的主观判断。

有趣的是，实际上导致应用程序性能被质疑的大部分情形都可以归纳为下面的少数几类。

- 用户不满意应用程序当前的性能表现。
- 系统监控工具警告某个基础组件正遭遇超时或不寻常的负载。
- 响应时间监控工具通知你某个服务级别协议没有得到满足。

第二点和第三点的区别尤为重要。基于此，接下来的两节将简要描述这些监控方案。之后再来看一下某些看似有必要、实则没必要进行优化的情况。

1.3.1 系统监控

系统监控工具基于一般系统统计信息进行健康检查。其目的是识别出不寻常的负载模式以及故障。虽然这些工具可以同时监控整个基础设施，但这里需要强调的是，它们只监控单独的组件（例如主机、应用服务器、数据库或存储子系统），而不考虑组件间的作用关系。因此，对于拥有复杂基础设施的环境，在支撑基础结构的单个组件出现异常时，很难或者几乎不可能评估异常对系统响应时间的影响。其中一个例子就是高频度地使用某一资源。换句话说，系统监控工具发出警报只是说明应用程序或者基础设施中的某些组件可能出现了问题，而用户根本没有察觉到任何性能问题（称作*误报*）；反之，也可能出现用户正在遭遇性能问题，而系统监控工具并没有发现问题（称作*漏报*）。最常见、也是最简单的关于误报和漏报的例子，是监控一个拥有许多CPU的对称多处理系统的CPU负载情况。假设你有一个装有四颗四核CPU的系统。当看到使用率在75%左右时，你可能觉得这太高了，系统受到了CPU的限制。但是，如果执行任务的数量远大于处理核心数量，那么这样的负载就是很正常的。这便是一个误报。反之，当你看到CPU使用率大约为8%时，你可能觉得一切正常。但是如果系统正在执行一个没有并行的单任务，那对于这个任务来说可能瓶颈就是CPU。实际上，100%的1/16只有6.25%，因此单个任务的CPU使用率不能超过6.25%。这便是一个漏报。

1.3.2 响应时间监控

响应时间监控工具（也称为*应用程序监控工具*）基于由机器人产生的假想事务或者由终端用户产生的真实事务进行监控。这些工具测量应用程序处理关键事务的时间，如果时间超出预期阈值，它们就会发出警告。换句话说，它们和用户一样利用基础设施，也会像用户那样“抱怨”糟糕的性能。因为它们从用户的角度监控应用程序，所以这些工具不仅能检查单个组件，更重要的是，它们还能检查整个应用程序的基础设施。因此它们专门用于监控服务级别协议。

1.3.3 强迫性调优障碍

曾经有一段时间，大部分数据库管理员都患上一种叫作“强迫性调优障碍”[①]的病症。其症状是过多地检查性能相关的统计信息（大部分都是基于比率的），从而无法集中精力关注真正重要的事情。他们简单地以为应用某些“简单”规则，就能优化所管理的数据库。历史告诉我们，结果并不总是尽如人意。为什么会出现这样的情况？所有用来检查某个给定比率（或值）的规则都是独立于用户体验而制定的。也就是说，误报和漏报都是规则而非意外。更糟糕的是，大量的时间消耗在这些任务上。

举例来说，时不时会有数据库管理员向我提出这样的问题：“我注意到我们的一个数据库在某个闩（latch）上有大量的等待。怎么做才能减少或者最好能消除这些等待？”一般我会回答：“你的用户因为这个闩锁抱怨过么？肯定没有，所以不用担心。反倒应该问问他们认为应用程序的问题有哪些。然后分析这些问题，你就会知道这个闩锁上的等待到底有没有影响到用户。”在下一节我会详细说明这个问题。

① 这个绝妙的名词是由Gaya Krishna Vaidyanatha发明的。你可以在*Oracle Insights: Tales of the Oak Table*（Apress，2004）一书中找到它的相关讨论。

虽然我从未做过数据库管理员，但是我必须承认我也患过“强迫性调优障碍”。现在我和其他大多数人一样，克服了这个病症。只不过与其他恶疾一样，彻底治愈“强迫性调优障碍”需要花很长时间。有些人根本没有意识到患了这种病症；有些人意识到了，但是因为多年的沉溺，总是很难去认识这样一个大错误并改掉陋习。

1.4 如何处理性能问题

简言之，应用程序的目标是向使用它的业务提供便利。因此优化应用程序性能的原因就是最大化这种便利。这并不意味着最大化性能，而是要在成本和性能之间找到最佳平衡点。事实上，优化任务中所投入的努力应该总能在预期回报中得到补偿。这意味着从业务视角来看，性能优化并非总有意义。

1.4.1 业务视角和系统视角

我们要优化一个为业务提供便利的应用程序的性能，所以当处理性能问题时，在深入应用程序的细节之前，必须要理解业务问题和需求。图1-2列出了拥有业务视角的人（即用户）和拥有系统视角的人（即工程师）之间的典型区别。

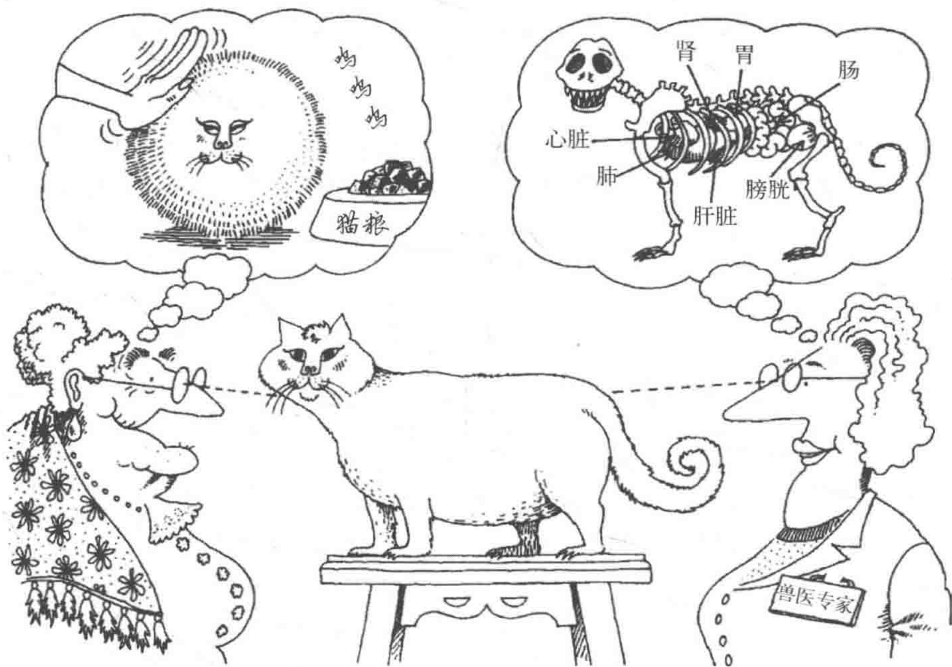


图1-2 不同的观察者可能会有完全不同的视角[①]

① 出自Grady Booch的著作*Object-Oriented Analysis and Design with Applications*第42页。引用已获得Grady Booch的许可。版权所有。

理解两个视角之间的因果关系非常重要。虽然要通过业务视角来理解结果，原因却需要从系统的视角来查看。所以，如果不想诊断不存在的或者不相关的问题（“强迫性调优障碍”），那么从业务视角来理解问题所在就非常重要，尽管这需要更精细的工作。

1.4.2　问题的编录

处理性能问题的第一步是从业务视角识别它们，并为其中的每一个问题都设定优先级和目标，如图1-3所示。

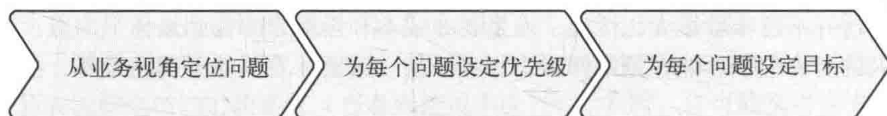


图1-3　编录性能问题时要完成的任务

业务问题无法通过系统视角发现。这些问题必须从业务视角识别。如果对服务级别协议的监控工作正常，则很容易通过查看不满足预期的操作来识别性能问题。否则，除了与用户或应用程序负责人交谈以外别无他法。这样的讨论会引出一系列的操作，比如注册新用户、运行报表或加载被认为缓慢的一堆数据。

警告　并非总是有必要从业务视角识别问题，有时候问题是已知的。例如，当有人告诉你下面这样的事情时，问题是需要识别的：“终端用户经常抱怨性能问题，请找出是什么原因导致的。”但是如果客户告诉你“运行某某报表花费的时间太长”，则无需额外的识别工作。对于后者，你已经知道要去检查应用程序的哪个部分了；而对于前者，你完全没有头绪，它可能牵扯应用程序的任何一个部分。

一旦识别出有问题的操作，就可以给它们分配优先级了。这就需要提出这样的问题：“如果只能解决五个问题，应该处理哪些呢？”当然，最好是能解决全部的问题，但是有时候时间和预算都有限。此外，还应考虑用于解决不同问题的方法相互冲突的情况。需要特别指出的是，在考虑优先级时，当前的性能表现可能是无关紧要的。例如，如果你正在处理一堆报表，并非一定是最慢的那个享有最高的优先级。可能最快的那个报表同时也是执行最频繁的，或者说是业务（干脆说是CEO）最关心的。这张报表可能因此拥有最高优先级而应当首先优化。再说一次，业务需求驱动你进行优化。

对于每个问题，都应该设置可量化的调优目标，例如“当单击创建用户的按钮之后，处理时间最多两秒”。如果性能需求甚至服务级别协议是可用的，那有可能优化目标是已知的。否则，再次强调，你必须考虑业务需求来制定目标。注意，没有目标意味着不知道何时停止寻找更好的方案。换句话说，调优过程将永无止境。记住，投入和产出要平衡。

1.4.3　解决问题

同时诊断多个问题比诊断单个问题更加复杂。因此应该尽量每次只解决一个问题。只要简单地根据问题列表的优先级顺序逐一检查就可以。

对于每个问题，都应该解答以下三个问题（如图1-4所示）。

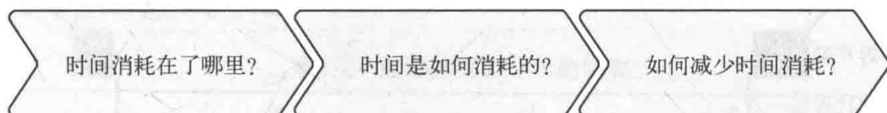


图1-4 要诊断一个性能问题，你应该解答这三个问题

- **时间消耗到哪里啦？** 首先，你必须找出时间都去哪儿啦。例如，一个具体的操作花费10秒钟，你必须找出这10秒钟内哪个模块或组件占用的时间最多。
- **时间是如何消耗的？** 一旦知道时间消耗到哪些地方，就得找出时间是如何消耗的。例如，你可能发现一个组件消耗了4.2秒在CPU上，用0.4秒的时间做磁盘I/O操作，用5.1秒的时间等待来自另一个组件的出队列消息。
- **如何减少时间消耗？** 最后，是时候找出如何让操作提速的办法了。为此，将精力集中在最消耗时间的部分十分关键。例如，如果磁盘I/O操作只占整个处理时间的4%，那么开始优化这些操作是没有意义的，即使这些操作运行非常缓慢。

要想找出时间消耗到哪些地方以及是如何消耗的，需要从收集你所关心的操作的端到端性能数据开始。这一点很关键，因为如今开发需要使用如Oracle这样的数据库应用程序，多层架构已成为事实标准。最简单的情况下，至少应实现两层（也称客户端/服务端）架构。大多数时候是三层架构：展现层、逻辑层和数据层。图1-5展示了部署Web应用程序的典型结构。出于安全或负载管理的目的，也常常会将组件分布在多台计算机上。



图1-5 一个典型Web应用由部署在多个系统上的多个组件构成

在多层架构中，请求的处理可能涉及多个组件。但是，未必在所有情况下处理某一请求时都涉及所有的组件。例如，如果激活了Web服务器级别的缓存，一个请求可能只在Web服务器端响应，而不需要发送至应用服务器端。当然同样的规则也适用于应用服务器或者数据库服务器。

理想情况下，要完整分析一个性能问题，应该收集处理过程中涉及的所有组件的详细信息。某些情形下，尤其是涉及许多组件时，也许有必要收集大量数据，这可能需要大量的时间来分析它们。基于这个原因，通常只有分步解决方案才是处理问题的唯一有效[1]途径。分步解决方案的思路是将端到端响应时间拆分到它的主要组件中去，以此作为分析的开始，然后在必要时才开始收集详细信息。也就是说，为了定位性能问题，应该只收集必要的、最少量的数据。

① 处理性能问题时，不仅应该优化正在分析的问题，同时应优化操作。也就是说，应该尽可能快地定位和修复问题。

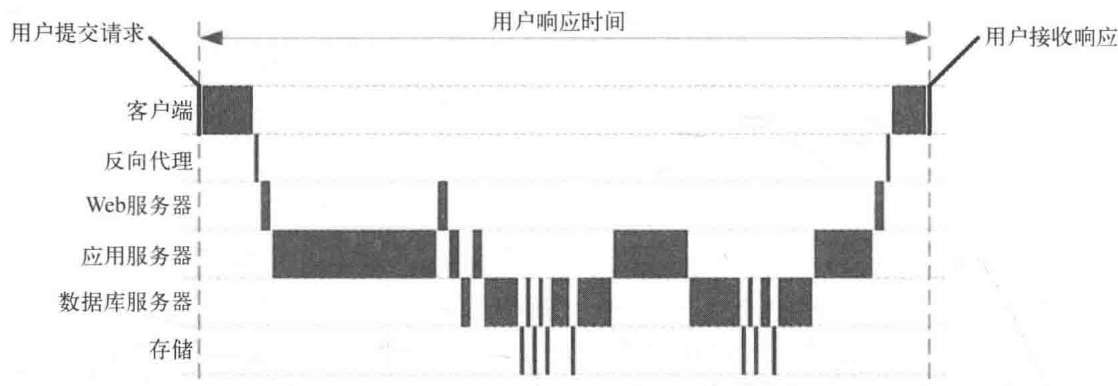


图1-6　将一个请求的响应时间拆分到所有主要组件中。组件间的通信延迟已忽略

一旦知道了涉及哪些组件以及每个组件各自消耗的时间，就可以进一步有选择性地收集那些最耗时间的组件的附加信息，从而分析问题所在。例如，根据图1-6所示，你只需考虑应用服务器和数据库服务器。完整分析那些只占响应时间一小部分的组件是毫无意义的。

根据用来收集性能数据的工具或技术的不同，很多情况下可能无法完全将响应时间拆分至如图1-6所示的每个组件中。况且，通常也没必要这样做。实际上，即使是像图1-7所示的局部分析，也能帮助确定是哪些组件消耗了大部分的响应时间。

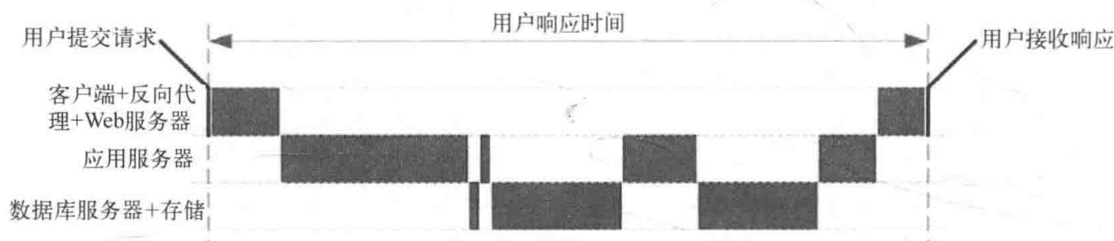


图1-7　一个请求的响应时间按组件进行部分分解

要收集与问题相关的性能数据，基本上只有以下两种可用方法。

- **检测**：如果一个应用程序的开发过程比较合理，那么它一定包含检测代码（instrumentation code），可以提供性能指标等信息。通常情况下，检测代码处于禁用状态，或者其输出维持最小化以节省资源。但是在应用程序运行时，应该可以激活检测代码或提供更多的信息量。Oracle的SQL跟踪（参见第3章）就是一个很好的例子。它默认是禁用的，但激活之后，却可以提供包含SQL语句执行的更多细节的跟踪文件。
- **探查分析**：探查器是一种性能分析工具，为运行中的应用程序记录执行的操作和执行它们所花费的时间，以及系统资源的使用情况（例如CPU和内存）。一些探查器在调用层面收集数据，而其他探查器则在代码层面收集数据。性能数据的收集或是通过按指定间隔来抓取应用程序状态，或是通过自动运行的检测代码或可执行文件进行。尽管前者的开销更低一些，但是通过后者收集的数据更精确。

通常来讲，调查性能问题时两种方法都会用到。但是，如果有好的检测可用，探查分析则会较少使用。表1-2总结了这两种技术的利弊。

表1-2 检测和探查分析的利弊

技术	优势	劣势
检测	可以向关键业务操作添加计时信息；当可用时，可被动态激活而不需部署新代码；上下文信息可用（例如关于用户和会话的信息）	必须手工实现；仅涵盖单个组件；没有端到端的响应时间视图；通常输出的格式取决于编写检测代码的开发者
探查分析	对于整个应用程序总是可用；多层探查器提供端到端的响应时间视图	可能非常昂贵，尤其是多层探查器；不能总是（快速地）部署在生产环境；在代码层面工作的相关负载可能会很高

不用说，只有当应用程序包含检测代码时才可以对其加以利用。然而，在某些情形以及实际中的很多时候，探查分析经常是唯一选择。

在开始解决某个特定问题时，值得注意的是，有时多亏副作用的影响而使其他问题得以修复（例如，减少CPU的使用可能让其他CPU敏感的操作获益，使得它们的运行趋于正常）。当然了，不好的一面也有可能发生。采取应对措施可能引入新的问题。因此仔细考虑修复指定问题时可能带来的所有副作用十分重要。同时也必须谨慎评估引入一个修复所含的固有风险。无疑所有的变更都应该在应用到生产环境之前进行仔细测试。

需要注意的是，在生产环境中问题的解决顺序并不总是依照优先级。有些措施可能需要花费更长的时间来实现。举例来说，变更一个高优先级的问题也许需要停机时间或者应用级别的修改。结果就是尽管有一些措施可以立即实现，而其他措施则可能需要几个星期或几个月甚至更长的时间来实现。

1.5 小结

本章描述了面对性能问题时的关键问题：为什么在正确的时间使用正确的方法处理性能问题是至关重要的，为什么一定要理解业务需求和问题所在，以及为什么有必要就良好性能（good performance）的含义达成一致。

在描述如何回答图1-4中的三个问题之前，需要介绍在本书剩余部分提到的几个关键概念。基于此，第2章将描述数据库引擎在执行SQL语句的过程中执行的操作。另外，也会提及检测的相关知识并给出几个常用术语的定义。

第2章

关键概念

本章的学习目标分为三个部分。首先，为避免不必要的困惑，我会介绍一些贯穿全书的术语，其中最重要的包括选择率、基数、游标、软解析、硬解析、绑定变量扫视以及自适应游标共享。其次，我会描述SQL语句的生命周期。换句话说，我会介绍为了执行SQL语句所涉及的操作。在讨论过程中，会重点关注解析。最后，我会描述如何检测应用程序代码和数据库调用。

2.1 选择率和基数

选择率（selectivity）是一个介于0和1之间的值，用来表示某个操作所返回的记录数的比例。例如，一个操作从表中读取120行，在应用过滤条件后，返回其中的18行，那么选择率就是0.15（18/120）。选择率也可以用百分比来表示，所以0.15也可以表示成15%。当选择率接近于0时，称之为具有强选择性；当选择率接近于1时，称之为具有弱选择性。

警告 我以前经常使用低/高或者好/坏这样的词表示强/弱的意思。之所以现在不再使用低/高，是因为这样的词无法明确表达其指的是选择率的程度高低还是其数值的高低。事实上，存在各种各样自相矛盾的定义。我不再使用好/坏是因为将质量的优劣与选择率联系在一起是不合理的。

一个操作返回记录的行数称作基数（cardinality）。公式2-1解释了选择率与基数之间的关系。

$$\text{基数} = \text{选择率} \times \text{行数}$$

公式2-1 选择率和基数之间的关系

警告 在关系模型中，基数指关系中的元组数量。因为当关系是一元的时候绝对不包含重复记录，元组的数量对应着其代表的不重复值的数量。可能基于这个原因，在一些出版物中，基数指的是某列中不重复的值的数量。因为SQL允许表中包含重复记录（也就是说SQL在这一点上并不遵守关系模型），我从不用基数表示某列中不重复的值的数量。另外，Oracle自身在定义这个术语时也并非完全一致。有时，Oracle在文档中用它来指不重复的值的数量，有时也用它来指一个操作返回的记录行数。

来看几个selectivity.sql脚本的例子。在下面的查询中，访问表的操作选择率是1。这是因为没有应用WHERE条件，因此查询返回了表中的所有记录。基数就等于表中的记录的行数，即10 000。

```
SQL> SELECT * FROM t;

...

10000 rows selected.
```

下面的查询中，访问表的操作基数是2601，因此选择率就是0.2601（返回10 000行中的2601行）。

```
SQL> SELECT * FROM t WHERE n1 BETWEEN 6000 AND 7000;

...

2601 rows selected.
```

下面的查询中，访问表的操作基数是0，因此选择率也是0（返回10 000行中的0行）。

```
SQL> SELECT * FROM t WHERE n1 = 19;

no rows selected.
```

在上面的三个例子中，与访问表操作相关的选择率是用查询表返回的基数除以表中存储的记录行数计算得来的。这种算法之所以可行，是因为三个查询都不包含连接或聚合操作。一旦查询中包含GROUP BY条件或者SELECT中含有聚合函数，则执行计划中至少应包含一个聚合操作。下面的查询说明了这一点（注意sum聚合函数）。

```
SQL> SELECT sum(n2) FROM t WHERE n1 BETWEEN 6000 AND 7000;

   SUM(N2)
----------
     70846

1 row selected.
```

在这类情形下，无法通过查询的基数（本例中为1）计算访问操作的选择率，而是应该通过类似下面的查询找出访问操作返回了多少行作为聚合函数的输入。此时，访问表的访问操作的基数是2601，因此选择率是0.2601（2601/10 000）。

```
SQL> SELECT count(*) FROM t WHERE n1 BETWEEN 6000 AND 7000;

  COUNT(*)
----------
      2601

1 row selected.
```

接下来你会发现（尤其是在第13章），了解一个操作的选择率有助于找到最高效的访问路径。

2.2 什么是游标

游标是指向私有SQL区（private SQL area）及其关联的共享SQL区（shared SQL area）的句柄（handle，一种允许程序访问某一资源的内存结构）。如图2-1所示，尽管句柄是客户端内存结构，但它指向了服务器进程的内存结构，转而指向存储在SGA中的内存结构，更确切地说是库缓存中的内存。

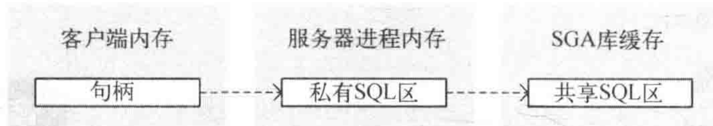


图2-1　游标是指向私有SQL区及其关联的共享SQL区的句柄

私有SQL区存储诸如绑定变量值和查询执行状态信息等数据。从命名上就可以看出，私有SQL区属于具体的会话。用于存储私有SQL区的会话内存称作用户全局区（UGA）。

共享SQL区包含两个独立的结构，即所谓的父游标（parent cursor）和子游标（child cursor）。存储在父游标中的关键信息是与游标关联的SQL语句文本，简单来说就是进程将要执行的SQL语句。存储在子游标中的关键元素是执行环境和执行计划。这些元素指明了执行过程如何进行。一个共享SQL区可以用于多个会话，因此它存储在库缓存中。

注意　在实践中，游标和私有/共享SQL区这两个术语可互换使用。

2.3　游标的生命周期

深入理解游标的生命周期是优化应用程序中SQL语句的必备知识。下面是处理游标过程中执行的步骤。

(1) 打开游标：在会话的UGA中会分配一个用于打开游标的私有SQL区。同时还会分配一个引用私有SQL区的客户端句柄。注意此时还没有任何SQL语句与该游标相关联。

(2) 解析游标：共享SQL区包含与该SQL语句解析后相关的表示形式及其执行计划（用来描述SQL引擎如何执行SQL语句），这些都是在SGA中生成和加载的，确切地说是在库缓存中。私有SQL区会进行更新，以存储一个对共享SQL区的引用。（下一节将讨论关于解析的更多内容。）

(3) 定义输出变量：如果SQL语句返回数据，则必须定义接收数据的变量。这不仅对查询是必要的，同样适用于使用了`RETURNING`条件的删除、插入和更新语句。

(4) 绑定输入变量：如果SQL语句使用了绑定变量，则必须为绑定变量提供值。在绑定过程中不执行检查。如果传入了非法数据，在执行的时候会抛出一个运行时错误。

(5) 执行游标：会在此阶段执行SQL语句。但是请注意，数据库引擎在这个阶段并不总是做些重要的事情。实际上，对于许多类型的查询，真正的处理过程都会推迟到获取阶段再做。

(6) 获取游标：如果SQL语句有结果，就在这一步取回结果。尤其对查询而言，这一阶段会执行大部分的处理过程。查询的时候可能只取回结果集的一部分，换句话说，游标可能在获取全部数据之前就关闭了。

(7) 关闭游标：与句柄和私有SQL区有关的资源被释放并可以供其他游标使用。库缓存中的共享SQL区则没有变化。它留在内存中希望以后可以被重用。

为了更好地理解这个过程，最好是按顺序单独思考图2-2中执行的每一步。不过在实践中，会使用不同的优化技巧来加速处理过程。例如，绑定变量扫视需要将执行计划的生成推迟到绑定变量的值变成已知的时候。

根据你所使用的编程环境或技术，图2-2中描述的不同步骤可能会被显式执行或隐式执行。为明确不同点，看一下图2-2下面两段来自lifecycle.sql脚本的PL/SQL代码块。两者有相同的目的（从emp表中读取一行），但是编码采用完全不同的方式。

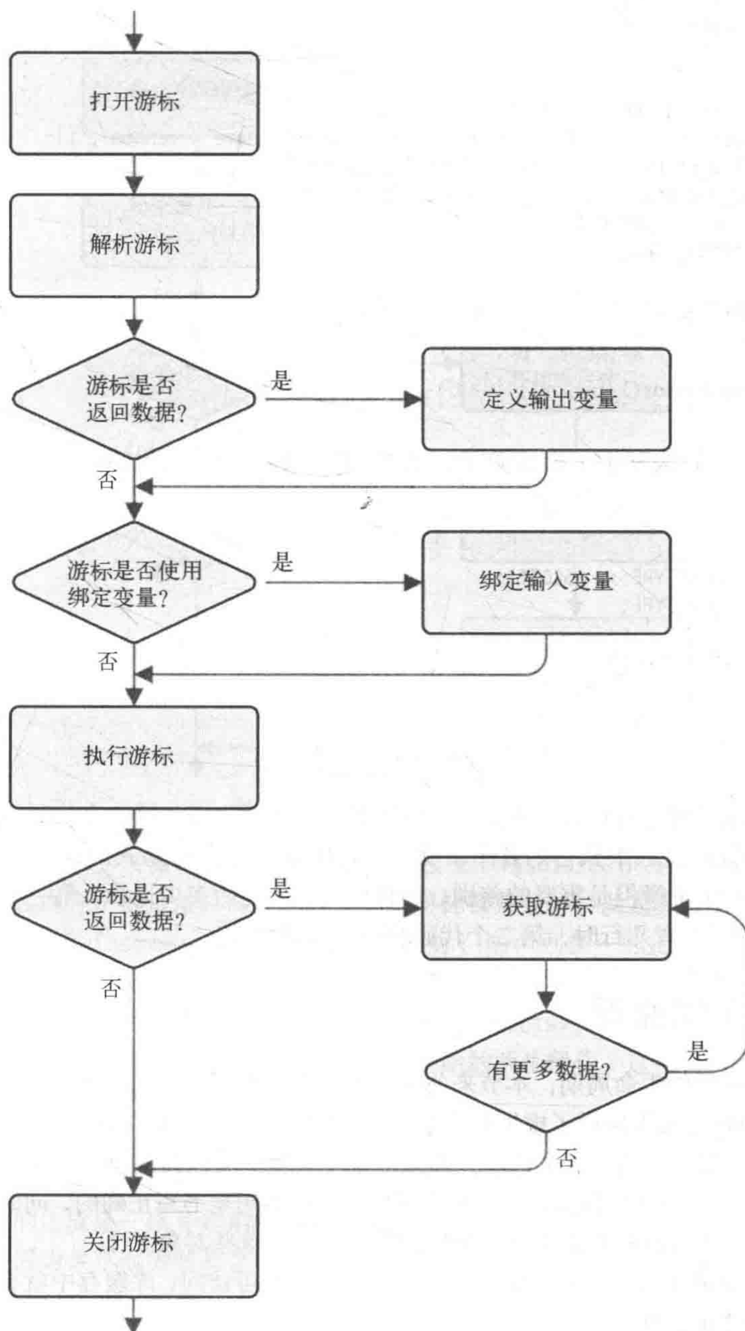


图2-2 游标的生命周期

第一个PL/SQL块使用dbms_sql包将图2-2中的每一步进行显式编码。

```
DECLARE
  l_ename emp.ename%TYPE := 'SCOTT';
  l_empno emp.empno%TYPE;
  l_cursor INTEGER;
  l_retval INTEGER;
BEGIN
  l_cursor := dbms_sql.open_cursor;
  dbms_sql.parse(l_cursor, 'SELECT empno FROM emp WHERE ename = :ename', 1);
  dbms_sql.define_column(l_cursor, 1, l_empno);
  dbms_sql.bind_variable(l_cursor, ':ename', l_ename);
  l_retval := dbms_sql.execute(l_cursor);
  IF dbms_sql.fetch_rows(l_cursor) > 0
  THEN
    dbms_sql.column_value(l_cursor, 1, l_empno);
    dbms_output.put_line(l_empno);
  END IF;
  dbms_sql.close_cursor(l_cursor);
END;
```

第二个PL/SQL代码块利用了隐式游标；基本上这个PL/SQL代码块将对游标的控制全权委托给PL/SQL编译器了。

```
DECLARE
  l_ename emp.ename%TYPE := 'SCOTT';
  l_empno emp.empno%TYPE;
BEGIN
  SELECT empno INTO l_empno
  FROM emp
  WHERE ename = l_ename;
  dbms_output.put_line(l_empno);
END;
```

多数的时间里编译器运行良好。事实上，编译器会在内部生成与第一个代码块类似的编码。但有时需要更多地控制处理过程中执行的各个步骤，因此不能总是使用隐式游标。举例来说，在这两个PLSQL块之间，有一个细微但是重要的差别。不管查询最终返回多少记录，第一个代码块不会产生异常。而当查询返回0行或者几行时，第二个代码块会产生异常。

2.4　解析的工作原理

上一节描述了游标的生命周期，本节来关注一下解析。如图2-3所示，解析执行的步骤如下。

(1) *包含VPD谓词*：如果使用了虚拟私有数据库（VPD，以前也称作行级别安全控制），并且解析的SQL语句其中的一张表激活了这个选项，那么由安全策略生成的谓词就会包含在WHERE条件中。

(2) *检查语法、语义和访问权限*：这一步不仅保证SQL语句是书写正确的，同时确保SQL语句引用的所有对象都存在，而且解析它的用户有相应的权限来访问这些对象。

(3) *在共享SQL区存储父游标*：只要可共享的父游标尚不可访问，库缓存中就会分配一些内存，新产生的父游标就存储在这里。

(4) *生成执行计划*：在这一阶段，查询优化器为解析的SQL语句产生执行计划（这个话题会在第6

章详细讨论)。

(5) 在共享SQL区存储子游标：此时会分配一些内存，可共享的子游标就存储在其中并与它的父游标进行关联。

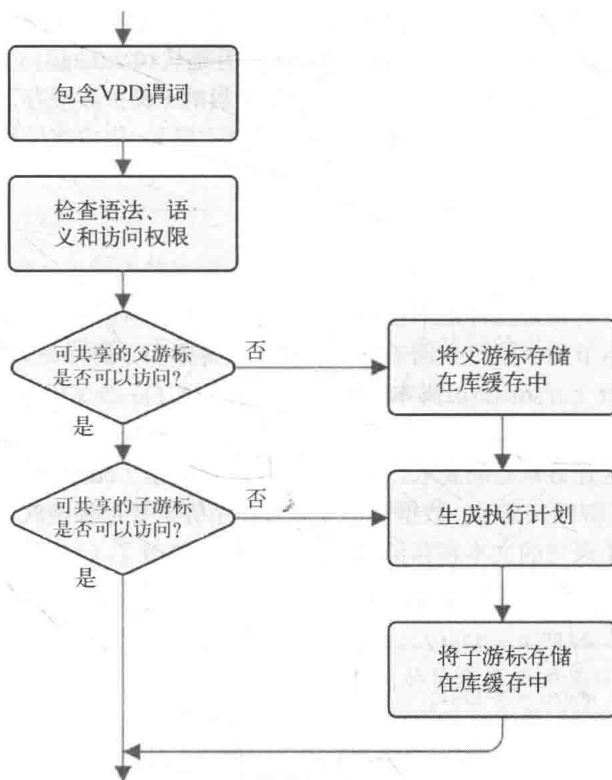


图2-3 解析阶段执行的步骤

一旦存储在库缓存中，父游标和子游标就分别通过视图`v$sqlarea`和`v$sql`具体化了。严格来讲，游标的标识符是其在内存中的地址，对于父游标和子游标都是这样。但大多数情况下，游标通过两个列值定位：`sql_id`和`child_number`。`sql_id`列定位父游标；两个列在一起定位子游标。但是也有例子表明这两列的值有时不足以定位一个游标。实际上，在有些版本[①]中，拥有许多子游标的父游标被废弃了并由新的父游标取代。因此，定位一个游标时还需要`address`列。

当存在可共享的父游标和子游标时，只需要执行开始的两步操作，这种解析称之为*软解析*；反之需要执行所有的操作时，称之为*硬解析*。

从性能的观点来看，应该尽可能地避免硬解析。这恰恰是数据库引擎在库缓存中存储共享游标的原因。这样一来，属于这个实例的每个进程都有可能重用它们。应该尽量避免硬解析的原因有两个：首先就是执行计划的生成是一项非常消耗CPU的操作；其次是存储在库缓存中的父游标和子游标需要共享池中的内存。因为共享池为所有会话所共享，所以共享池的内存分配是串行的。基于这个目的，

① 每个父游标可以拥有的最大子游标数量经历了几次变更：截至11.1.0.6版本，该数字是1026个；从11.1.0.7到11.2.0.1版本，是32 768个；在11.2.0.2中是65 536个；截至11.2.0.3版本，是100个。

一个用于保护共享池的闩（shared pool latch）必须获取分配给父游标和子游标的内存。因为串行化的原因，引发大量硬解析的应用程序有可能正在遭遇共享池的闩的竞争。尽管软解析比硬解析的影响要低得多，但是因为软解析同样受到序列化的限制，所以避免软解析也同样重要。实际上，数据库引擎必须保证在搜寻可共享的游标时所访问的内存结构不被修改。真实的实现取决于不同的版本：10.2.0.1版本必须获得一个属于库缓存的闩（library cache latch），但是从10.2.0.2起，Oracle开始用互斥（mutex）替代库缓存的闩，而到了11.1版本时则只有互斥用于这个目的（保护库缓存）。总之，考虑到软解析和硬解析对应用程序可扩展性的限制（第12章会详细讨论该主题），你应该尽量避免它们出现。

2.4.1 可共享游标

解析操作的结果就是一个父游标和一个子游标存储在库缓存中的共享SQL区中。显然，在共享的内存区域中存储它们的目的就是允许重用它们从而避免硬解析。所以有必要讨论一下哪些情况下可以重用父游标或子游标。本节列举了三个例子来说明共享父游标和子游标是如何运作的。

基于sharable_parent_cursors.sql脚本的第一个例子展示了一个父游标在哪里不能进行共享。与父游标相关的关键信息是SQL语句的文本。因此，一般而言，如果几个SQL语句的文本完全一样才可以共享同一个父游标。这是最核心的要求。但是当启用游标共享（cursor sharing）时也会出现例外的情况。实际上，当启用了游标共享时，数据库引擎会自动用绑定变量替换SQL语句中的字面值。因此，数据库引擎接收到的SQL语句的文本在存储到父游标之前被修改了（详见第12章）。第一个例子中使用了四个SQL语句，其中有两个SQL语句有相同的文本，另外两个只是字母大小写或者空格不一样。

```
SQL> SELECT * FROM t WHERE n = 1234;

SQL> select * from t where n = 1234;

SQL> SELECT   *   FROM    t  WHERE   n=1234;

SQL> SELECT * FROM t WHERE n = 1234;
```

通过v$sqlarea视图，可以确定创建了三个不同的父游标。同时注意每个游标执行的次数。

```
SQL> SELECT sql_id, sql_text, executions
  2  FROM v$sqlarea
  3  WHERE sql_text LIKE '%1234';

SQL_ID        SQL_TEXT                                 EXECUTIONS
------------- ---------------------------------------- ----------
2254m1487jg50 select * from t where n = 1234                    1
g9y3jtp6ru4cb SELECT * FROM t WHERE n = 1234                    2
7n8p5s2udfdsn SELECT   *   FROM    t   WHERE   n=1234           1
```

第二个例子的目的是，通过sharable_child_cursors.sql脚本展示父游标可以共享而子游标不能共享。与子游标相关的关键信息是执行计划和相关执行环境。因而，几个SQL语句能够共享子游标的条件是它们拥有同一个父游标并且执行环境是相互兼容的。为说明这一点，在给初始化参数optimizer_mode设置两个不同值的情况下执行同一个SQL语句。

```
SQL> ALTER SESSION SET optimizer_mode = all_rows;
```

```
SQL> SELECT count(*) FROM t;

COUNT(*)
--------
    1000

SQL> ALTER SESSION SET optimizer_mode = first_rows_1;

SQL> SELECT count(*) FROM t;

COUNT(*)
--------
    1000
```

执行的结果是创建了一个单独的父游标（5tjqf7sx5dzmj）和两个子游标（0和1）。同时还要注意到两个子游标都拥有相同的执行计划（plan_hash_value列的值相同）。这很好地证明了创建新的子游标是因为新的截然不同的执行环境，而不是因为生成了另一个执行计划。

```
SQL> SELECT sql_id, child_number, optimizer_mode, plan_hash_value
  2  FROM v$sql
  3  WHERE sql_text = 'SELECT count(*) FROM t';

SQL_ID        CHILD_NUMBER OPTIMIZER_MODE PLAN_HASH_VALUE
------------- ------------ -------------- ---------------
5tjqf7sx5dzmj            0 ALL_ROWS            2966233522
5tjqf7sx5dzmj            1 FIRST_ROWS          2966233522
```

警告 如上面的例子所示，1号子游标optimizer_mode列的值没有正确显示。实际上，该列显示的是FIRST_ROWS，而不是FIRST_ROWS_1。同样的行为也可以在使用FIRST_ROWS_10、FIRST_ROWS_100和FIRST_ROWS_1000时观察到。这可能会导致潜在的问题：即使执行环境不一样，SQL引擎也不区分其中的不同。因此，可能会错误地共享子游标。

要想知道哪些不匹配导致出现了几个子游标，可以查询v$sql_shared_cursor视图。在这个视图中你可能会发现，对于每个子游标（除了第一个编号为0的），都会显示为何不能共享之前建立的子游标。对于几种类型的不一致（在12.1版本中是64个），都有一列将值设置为N（没有不匹配）或Y（不匹配）。通过下面的查询，可以确认在之前的例子中，第二个子游标的不匹配是由于不同的优化器模式。

```
SQL> SELECT optimizer_mode_mismatch
  2  FROM v$sql_shared_cursor
  3  WHERE sql_id = '5tjqf7sx5dzmj'
  4  AND child_number = 1;

OPTIMIZER_MODE_MISMATCH
-----------------------
Y
```

在11.2.0.2中，v$sql_shared_cursor视图提供一个称作reason的列。该列不仅提供导致出现新的子游标不匹配的文本描述，而且提供不匹配的的额外信息。因为reason列包含的信息和不匹配的类型息息相关，所以它的类型是CLOB，并且它使用XML格式。例如，在接下来的示例中，三个XML元素包含

了关键信息。原因（Optimizer mismatch）存储在reason元素中，库缓存中已存在的游标的优化器模式（也就是1，表示ALL_ROWS）存储在optimizer_mode_cursor元素中，会话解析语句所需的优化器模式（2，即FIRST_ROWS）存储在optimizer_mode_current元素中。

```
SQL> SELECT reason
  2  FROM v$sql_shared_cursor
  3  WHERE sql_id = '5tjqf7sx5dzmj'
  4  AND child_number = 0;

REASON
--------------------------------------------------------------------------------------------
<ChildNode><ChildNumber>0</ChildNumber><ID>3</ID><reason>Optimizer mismatch(10)</reason><siz
e>3x4</size><optimizer_mode_hinted_cursor>0</optimizer_mode_hinted_cursor><optimizer_mode_cu
rsor>1</optimizer_mode_cursor><optimizer_mode_current>2</optimizer_mode_current></ChildNode>

SQL> SELECT x.reason,
  2         decode(x.optimizer_mode_cursor,
  3                1, 'ALL_ROWS',
  4                2, 'FIRST_ROWS',
  5                3, 'RULE',
  6                4, 'CHOOSE', x.optimizer_mode_cursor) AS optimizer_mode_cursor,
  7         decode(x.optimizer_mode_current,
  8                1, 'ALL_ROWS',
  9                2, 'FIRST_ROWS',
 10                3, 'RULE',
 11                4, 'CHOOSE', x.optimizer_mode_current) AS optimizer_mode_current
 12  FROM v$sql_shared_cursor s,
 13       XMLTable('/ChildNode'
 14                PASSING XMLType(reason)
 15                COLUMNS
 16                  reason VARCHAR2(100)          PATH '/ChildNode/reason',
 17                  optimizer_mode_cursor NUMBER  PATH '/ChildNode/optimizer_mode_cursor',
 18                  optimizer_mode_current NUMBER PATH '/ChildNode/optimizer_mode_current'
 19                     ) x
 20  WHERE s.sql_id = '5tjqf7sx5dzmj'
 21  AND s.child_number = 0;

REASON                 OPTIMIZER_MODE_CURSOR OPTIMIZER_MODE_CURRENT
---------------------- --------------------- ----------------------

Optimizer mismatch(10) ALL_ROWS              FIRST_ROWS
```

第三个例子仍然是基于sharable_child_cursors.sql脚本，目的是展示执行环境不仅影响执行计划，还有可能会影响SQL语句的结果。这也是共享子游标的执行环境必须相互兼容的另一个原因。例如，下面的SQL语句的输出证实了nls_sort初始化参数的影响。

```
SQL> ALTER SESSION SET nls_sort = binary;

SQL> SELECT * FROM t ORDER BY pad;

  N PAD
--- ---
  1 1
  2 =
```

```
  3 Z
  4 z

SQL> ALTER SESSION SET nls_sort = xgerman;

SQL> SELECT * FROM t ORDER BY pad;

  N PAD
--- ---
  2 =
  4 z
  3 Z
  1 1
```

因为执行环境的不同，使用了同一个父游标下的两个子游标。注意在这个案例中，不匹配可以通过v$sql_shared_cursor视图查看，特别是在language_mismatch列中。

```
SQL> SELECT sql_id, child_number, plan_hash_value, executions
  2  FROM v$sql
  3  WHERE sql_text = 'SELECT * FROM t ORDER BY pad';

SQL_ID        CHILD_NUMBER PLAN_HASH_VALUE EXECUTIONS
------------- ------------ --------------- ----------
1f7qg6nu40shd            0       961378228          1
1f7qg6nu40shd            1       961378228          1

SQL> SELECT child_number, language_mismatch
  2  FROM v$sql_shared_cursor
  3  WHERE sql_id = '1f7qg6nu40shd'
  4  AND child_number > 0;

CHILD_NUMBER LANGUAGE_MISMATCH
------------ -----------------
           1 Y
```

在实践中，由不可共享的父游标导致的硬解析远比由不可共享的子游标导致的硬解析更加常见。事实上，多半情况是因为每个父游标只有较少的子游标。如果父游标无法共享，通常是因为SQL语句的文本变更的结果。这多发于SQL语句由应用程序动态生成或用字面值替代了绑定变量的情况。一般来讲动态生成SQL语句无法避免。另一方面，通常都可以使用绑定变量。但是，并不是什么情况下都适合使用绑定变量。接下来关于绑定变量利弊的讨论，可以帮你理解什么时候使用它们是合适的，什么时候是不合适的。

2.4.2 绑定变量

绑定变量通过三种方式影响应用程序。第一，从开发角度来看，它们既可以让编程变简单，也可以让编程变复杂（更准确地说，就是需要编写的代码或多或少）。这种情况下，影响取决于用来执行SQL语句的应用编程接口。例如，如果你正在编写PL/SQL代码，使用绑定变量来执行会更容易。另一方面，如果你正在使用JDBC编写Java程序，没有绑定变量的情况下执行SQL语句会更容易。第二，从安全角度看，绑定变量减轻了SQL注入攻击的风险。第三，从性能角度看，使用绑定变量有利有弊。

注意　在接下来的小节里，你会看见一些执行计划。第10章将阐述如何获得和解释执行计划。如果你读完第10章后有什么不清楚的，可以考虑回过头来阅读本章。

1. 优势

绑定变量在性能方面的优势是它们允许共享库缓存中的父游标，这样就避免了硬解析以及相关的额外开销。接下来的例子是对脚本bind_variables_graduation.sql的输出的摘录，展示了三个INSERT语句由于使用绑定变量而共享了库缓存中的同一个游标。

```
SQL> VARIABLE n NUMBER

SQL> VARIABLE v VARCHAR2(32)

SQL> EXECUTE :n := 1; :v := 'Helicon';

SQL> INSERT INTO t (n, v) VALUES (:n, :v);

SQL> EXECUTE :n := 2; :v := 'Trantor';

SQL> INSERT INTO t (n, v) VALUES (:n, :v);

SQL> EXECUTE :n := 3; :v := 'Kalgan';

SQL> INSERT INTO t (n, v) VALUES (:n, :v);

SQL> SELECT sql_id, child_number, executions
  2  FROM v$sql
  3  WHERE sql_text = 'INSERT INTO t (n, v) VALUES (:n, :v)';

SQL_ID        CHILD_NUMBER EXECUTIONS
------------- ------------ ----------
6cvmu7dwnvxwj            0          3
```

但是有些情况下，即使使用了绑定变量，还是创建了几个子游标，如下面的例子所示。注意，INSERT语句和之前的例子是一样的。只是VARCHAR2变量的最大值发生了改变（从32到33）。

```
SQL> VARIABLE v VARCHAR2(33)

SQL> EXECUTE :n := 4; :v := 'Terminus';

SQL> INSERT INTO t (n, v) VALUES (:n, :v);

SQL> SELECT sql_id, child_number, executions
  2  FROM v$sql
  3  WHERE sql_text = 'INSERT INTO t (n, v) VALUES (:n, :v)';

SQL_ID        CHILD_NUMBER EXECUTIONS
------------- ------------ ----------
6cvmu7dwnvxwj            0          3
6cvmu7dwnvxwj            1          1
```

创建新的子游标（1）是因为前面三个INSERT语句和第四个之间的执行环境发生了改变。下面的例子中的不匹配项，可以通过查询v$sql_shared_cursor视图来确认。注意，bind_length_upgradeable列只在11.2版本中存在。在之前的版本中，这个信息由bind_mismatch列提供。

```
SQL> SELECT child_number, bind_length_upgradeable
  2  FROM v$sql_shared_cursor
  3  WHERE sql_id = '6cvmu7dwnvxwj';

CHILD_NUMBER BIND_LENGTH_UPGRADEABLE
------------ -----------------------
           0 N
           1 Y
```

这是因为数据库引擎使用了一个叫作绑定变量分级的特性。这个特性的目标是通过将绑定变量按等级（随大小变化）分成四个组来最小化子游标的数量。第一组包含最大至32字节的绑定变量，第二个组包含33至128字节的绑定变量，第三组包含大小为129至2000字节的绑定变量，最后一组包含大于2000字节的绑定变量。NUMBER数据类型的绑定变量按它们的最大长度22字节划分等级。如下面的例子所示，v$sql_bind_metadata视图显示了每个组的最大长度。注意值128的用法，即使子游标1的绑定变量长度定义为33。

```
SQL> SELECT s.child_number, m.position, m.max_length,
  2         decode(m.datatype,1,'VARCHAR2',2,'NUMBER',m.datatype) AS datatype
  3  FROM v$sql s, v$sql_bind_metadata m
  4  WHERE s.sql_id = '&sql_id'
  5  AND s.child_address = m.address
  6  ORDER BY 1, 2;

CHILD_NUMBER   POSITION MAX_LENGTH DATATYPE
------------ ---------- ---------- --------
           0          1         22 NUMBER
           0          2         32 VARCHAR2
           1          1         22 NUMBER
           1          2        128 VARCHAR2
```

注意 这个例子展示了当使用不同组的绑定变量时出现了绑定错配的情况。只有当关联到新的组的绑定变量比原来大时才会出现这种情况。实际上，仔细回顾这个例子，绑定变量的大小一直在增加。如果它们是在减小，那么所有的执行都可以共享同一个子游标。如果用VARCHAR2类型的最大值创建子游标，那么所有比它小的VARCHAR2绑定变量都可以共享它。

很显然，每次产生一个新的子游标就表示一个执行计划的生成。这个新的执行计划是否能够被其他子游标使用也取决于绑定变量的值。这部分内容将在下一节讨论。

2. 劣势

在WHERE条件中使用绑定变量对于性能方面的劣势是，在某些条件下会对查询优化器隐藏重要的信息。事实上，对于查询优化器而言，获取字面值比使用绑定变量更好。使用字面值时，查询优化器总能够做出最接近的估算。当涉及范围比较谓词（例如基于BETWEEN、大于或小于的比较条件），检查

一个值是否在可用值范围之外时（即小于列中存储的最小值或大于列中存储的最大值），或者使用直方图时，情况尤其如此。例如，拿一个1000行数据的表来说，在id列上，所有的整型值都在1（最小值）和1000（最大值）之间。

```
SQL> SELECT count(id), count(DISTINCT id), min(id), max(id) FROM t;

 COUNT(ID) COUNT(DISTINCTID)    MIN(ID)    MAX(ID)
---------- ----------------- ---------- ----------
      1000              1000          1       1000
```

当一个用户选择id小于990的所有记录时，查询优化器就知道（归功于对象统计信息）表中大约99%的数据被选中了。因此，它会选择使用全表扫描的执行计划。同时还要注意估算的基数（执行计划中的Rows列）几乎准确对应查询应返回的行数。

```
SQL> SELECT count(pad) FROM t WHERE id < 990;

COUNT(PAD)
----------
       989

-------------------------------------------
| Id  | Operation           | Name | Rows  |
-------------------------------------------
|   0 | SELECT STATEMENT    |      |       |
|   1 |  SORT AGGREGATE     |      |     1 |
|   2 |   TABLE ACCESS FULL| T    |   990 |
-------------------------------------------
```

当另一个用户选择id小于10的所有记录时，查询优化器知道表中仅有大约1%的数据被选中。因此，它选择使用索引扫描的执行计划。在这个例子中同样要注意其非常准确的估算。

```
SQL> SELECT count(pad) FROM t WHERE id < 10;

COUNT(PAD)
----------
         9
-----------------------------------------------------
| Id  | Operation                     | Name | Rows  |
-----------------------------------------------------
|   0 | SELECT STATEMENT              |      |       |
|   1 |  SORT AGGREGATE               |      |     1 |
|   2 |   TABLE ACCESS BY INDEX ROWID| T    |     9 |
|   3 |    INDEX RANGE SCAN           | T_PK |     9 |
-----------------------------------------------------
```

处理绑定变量时，查询优化器习惯于忽略它们的值。因此，像之前的例子中的完美估算是不可能的。为解决这个问题，Oracle9*i*中引入了一个叫作*绑定变量扫视*（bind variable peeking）的特性。绑定变量扫视的概念很简单：在生成执行计划之前，查询优化器扫视绑定变量的值并将其作为字面值使用。这个方法的问题在于执行计划的生成依赖于第一次执行所提供的值。下面这个基于bind_variables_peeking.sql脚本的例子就验证了这种行为。注意第一次优化是按照值990执行的。结果就是查询优化器选择全表扫描。正是这个选择，一旦游标被共享，就会影响使用值为10的第二个查询。

```
SQL> VARIABLE id NUMBER

SQL> EXECUTE :id := 990;

SQL> SELECT count(pad) FROM t WHERE id < :id;

COUNT(PAD)
----------
       989

-------------------------------------------
| Id | Operation          | Name | Rows   |
-------------------------------------------
|  0 | SELECT STATEMENT   |      |        |
|  1 |  SORT AGGREGATE    |      |      1 |
|  2 |   TABLE ACCESS FULL| T    |    990 |
-------------------------------------------

SQL> EXECUTE :id := 10;

SQL> SELECT count(pad) FROM t WHERE id < :id;

COUNT(PAD)
----------
         9

-------------------------------------------
| Id | Operation          | Name | Rows   |
-------------------------------------------
|  0 | SELECT STATEMENT   |      |        |
|  1 |  SORT AGGREGATE    |      |      1 |
|  2 |   TABLE ACCESS FULL| T    |    990 |
-------------------------------------------
```

当然，如下例所示，如果第一个执行换成值10，查询优化器就会选择使用索引扫描的执行计划，这意味着两个查询又一次都这样做了。注意，为避免和前一个例子共享游标，查询用小写字母来书写。

```
SQL> EXECUTE :id := 10;

SQL> select count(pad) from t where id < :id;

COUNT(PAD)
----------
         9

-----------------------------------------------------
| Id | Operation                    | Name | Rows   |
-----------------------------------------------------
|  0 | SELECT STATEMENT             |      |        |
|  1 |  SORT AGGREGATE              |      |      1 |
|  2 |   TABLE ACCESS BY INDEX ROWID| T    |      9 |
|  3 |    INDEX RANGE SCAN          | T_PK |      9 |
-----------------------------------------------------

SQL> EXECUTE :id := 990;
```

```
SQL> select count(pad) from t where id < :id;

COUNT(PAD)
----------
       989

--------------------------------------------------------
| Id | Operation                    | Name | Rows  |
--------------------------------------------------------
|  0 | SELECT STATEMENT             |      |       |
|  1 |  SORT AGGREGATE              |      |     1 |
|  2 |   TABLE ACCESS BY INDEX ROWID| T    |     9 |
|  3 |    INDEX RANGE SCAN          | T_PK |     9 |
--------------------------------------------------------
```

一定要理解，只要游标保留在库缓存中并可以共享，就会被重用。这和与其关联的执行计划的效率无关。

为解决这个问题，从11.1版本开始，数据库引擎启用一个称为*自适应游标共享*（adaptive cursor sharing，也称为*绑定感知游标共享*，bind-aware cursor sharing）的新特性。它的目的是自动识别出因重复利用已经可用的游标导致的低效的执行。要理解这个特性如何工作，我们从查看由v$sql提供的一些信息开始。下面是11.1版本中可用的新列。

- is_bind_sensitive不仅表明绑定变量扫视是否用于生成执行计划，同时也表示自适应游标共享可能会被考虑。如果是这样，此列值设置为Y，否则就设置为N。
- is_bind_aware表明游标是否使用自适应游标共享。如果是，列值为Y；如果不是，则设置为N。
- is_shareable表明游标是否可共享。如果可以，列设置为Y；否则，值为N。如果值为N，则游标不再被重用。

下面的例子来自于adaptive_cursor_sharing.sql脚本，游标是可共享的并且是绑定变量的，但并没有使用自适应游标共享。

```
SQL> EXECUTE :id := 10;

SQL> SELECT count(pad) FROM t WHERE id < :id;

COUNT(PAD)
----------
         9

SQL> SELECT sql_id
  2  FROM v$sqlarea
  3  WHERE sql_text = 'SELECT count(pad) FROM t WHERE id < :id';

SQL_ID
-------------
asth1mx10aygn

SQL> SELECT child_number, is_bind_sensitive, is_bind_aware, is_shareable, plan_hash_value
  2  FROM v$sql
  3  WHERE sql_id = 'asth1mx10aygn';
```

```
CHILD_NUMBER IS_BIND_SENSITIVE IS_BIND_AWARE IS_SHAREABLE PLAN_HASH_VALUE
------------ ----------------- ------------- ------------ ---------------
           0 Y                 N             Y                 4270555908
```

当游标使用不同的绑定变量值执行了几次后，有意思的事情发生了。注意下面编号为0的子游标不再是可共享的，并且两个新的子游标替换了它，它们都使用了自适应游标共享。

```
SQL> EXECUTE :id := 990;

SQL> SELECT count(pad) FROM t WHERE id < :id;

COUNT(PAD)
----------
       989

SQL> EXECUTE :id := 10;

SQL> SELECT count(pad) FROM t WHERE id < :id;

COUNT(PAD)
----------
         9

SQL> SELECT child_number, is_bind_sensitive, is_bind_aware, is_shareable, plan_hash_value
  2  FROM v$sql
  3  WHERE sql_id = 'asth1mx10aygn'
  4  ORDER BY child_number;

CHILD_NUMBER IS_BIND_SENSITIVE IS_BIND_AWARE IS_SHAREABLE PLAN_HASH_VALUE
------------ ----------------- ------------- ------------ ---------------
           0 Y                 N             N                 4270555908
           1 Y                 Y             Y                 2966233522
           2 Y                 Y             Y                 4270555908
```

查看与游标关联的执行计划，可能如你所期待的，你会看见其中一个新的子游标拥有基于全表扫描的执行计划，而另一个则基于索引扫描。

```
Plan hash value: 4270555908

---------------------------------------------
| Id | Operation                    | Name |
---------------------------------------------
|  0 | SELECT STATEMENT             |      |
|  1 |  SORT AGGREGATE              |      |
|  2 |   TABLE ACCESS BY INDEX ROWID| T    |
|  3 |    INDEX RANGE SCAN          | T_PK |
---------------------------------------------

Plan hash value: 2966233522

-----------------------------------
| Id | Operation          | Name |
-----------------------------------
|  0 | SELECT STATEMENT   |      |
|  1 |  SORT AGGREGATE    |      |
|  2 |   TABLE ACCESS FULL| T    |
-----------------------------------
```

要进一步分析两个新的子游标产生的原因，可以使用下面几个动态性能视图：v$sql_cs_statistics、v$sql_cs_selectivity和v$sql_cs_histogram。第一个视图表明是否使用了扫视以及与每个子游标相关的执行统计信息。在下面的输出中，可以确认对于一次执行，子游标1处理的行数比子游标2要高。这是查询优化器在一种情况下选择全表扫描而在另一种情况下选择索引扫描的主要原因。

```
SQL> SELECT child_number, peeked, executions, rows_processed, buffer_gets
  2  FROM v$sql_cs_statistics
  3  WHERE sql_id = 'asth1mx10aygn'
  4  ORDER BY child_number;

CHILD_NUMBER PEEKED EXECUTIONS ROWS_PROCESSED BUFFER_GETS
------------ ------ ---------- -------------- -----------
           0 Y               1             19           3
           1 Y               1            990          18
           2 Y               1             19           3
```

v$sql_cs_selectivity视图显示与每个子游标的每个谓词相关的选择率范围。实际上，数据库引擎并不会为每个绑定变量值创建一个新的子游标。相反，它将拥有大致相同的选择率的值分到同一个组，从而导致相同的执行计划。

```
SQL> SELECT child_number, trim(predicate) AS predicate, low, high
  2  FROM v$sql_cs_selectivity
  3  WHERE sql_id = 'asth1mx10aygn'
  4  ORDER BY child_number;

CHILD_NUMBER PREDICATE LOW        HIGH
------------ --------- ---------- ----------
           1 <ID       0.890991   1.088989
           2 <ID       0.008108   0.009910
```

v$sql_cs_selectivity视图的信息不仅用于展示每个子游标的选择率范围，而且数据库引擎也可使用该信息来选择使用哪个子游标。实际上，当一个游标是绑定感知的，绑定变量扫视会取代每一次的解析执行，而且游标的谓词选择率是基于估算的。根据这个估算选用正确的子游标。或者，如果没有适用于这个选择率范围的游标，则创建一个新的子游标。

警告　绑定感知的游标是必要的，对于每次解析，查询优化器都对它们的谓词进行选择率的估算。基于这个原因，数据库引擎有时会禁用自适应游标共享。有两个常见情况需要考虑：第一个是当SQL语句包含的绑定变量超过14个时；第二个是当查询优化器不能正确估算选择率时。例如，当变量需要隐式数据类型转换（这是使用正确数据类型的另一个理由），选择率无法估算出来时，或者引用的对象没有对象统计信息时。

---

v$sql_cs_histogram视图的内容由SQL引擎用来决定何时将一个游标置于绑定感知，以及应何时使用自适应游标共享。对于每一个子游标，这个视图会显示三个桶。第一个桶（bucket_id等于0）与高效的执行相关，第二个桶（bucket_id等于1）与低效的执行相关，第三个桶（bucket_id等于2）与效率非常低的执行相关。思路是：在完成一次执行后，SQL引擎比较估算的基数和实际的基数。然后，根据这两个基数有多接近，本次执行与三个桶中的一个相关联（换言之count列增加了）。稍后，当执

行涉及同一个游标的下一阶段操作时，以及涉及执行在这三个桶中间如何分布时，一个游标可能会变成绑定感知的或非感知的。举例来说，当低效的执行次数和高效执行次数一样多时，游标就被置为绑定感知的。接下来的例子证明了这点（注意，对于编号0的子游标，高效的执行次数和低效的执行次数相同）。

```
SQL> SELECT child_number, bucket_id, count
  2  FROM v$sql_cs_histogram
  3  WHERE sql_id = 'asth1mx10aygn'
  4  ORDER BY child_number, bucket_id;

CHILD_NUMBER  BUCKET_ID      COUNT
------------ ---------- ----------
           0          0          1
           0          1          1
           0          2          0
           1          0          1
           1          1          0
           1          2          0
           2          0          1
           2          1          0
           2          2          0
```

为了更好地理解如何使用v$sql_cs_histogram视图的内容，我建议你用adaptive_cursor_sharing_histogram.sql中的脚本做以下几种情况的实验。

自适应游标共享有两个主要的限制。第一，默认情况下，游标是按照绑定不敏感创建的。第二，对于给定的游标，绑定感知不是持续的。结果就是，在一个游标从自适应游标共享中获益之前，至少有一次执行是无效率的，在某些情况下甚至有多次执行（当曾经有很多次高效执行时）是无效率的。自11.1.0.7版开始，才有可能通过指定bind_aware这个hint[①]来避免这些限制。注意，在下面的例子中，两个子游标都是绑定敏感的，且都使用了高效的执行计划。

```
SQL> EXECUTE :id := 10;

SQL> SELECT /*+ bind_aware */ count(pad) FROM t WHERE id < :id;

COUNT(PAD)
----------
         9

Plan hash value: 4270555908

---------------------------------------------
| Id | Operation                    | Name |
---------------------------------------------
|  0 | SELECT STATEMENT             |      |
|  1 |  SORT AGGREGATE              |      |
|  2 |   TABLE ACCESS BY INDEX ROWID| T    |
|  3 |    INDEX RANGE SCAN          | T_PK |
---------------------------------------------

SQL> EXECUTE :id := 990;
```

① 这个词在书中未翻译，因为译为“提示”有时候会导致将其原有含义淹没在译文中。——译者注

```
SQL> SELECT /*+ bind_aware */ count(pad) FROM t WHERE id < :id;

COUNT(PAD)
----------
       989

Plan hash value: 2966233522

-----------------------------------
| Id | Operation           | Name |
-----------------------------------
|  0 | SELECT STATEMENT    |      |
|  1 |  SORT AGGREGATE     |      |
|  2 |   TABLE ACCESS FULL | T    |
-----------------------------------

SQL> SELECT child_number, is_bind_sensitive, is_bind_aware, is_shareable, plan_hash_value
  2  FROM v$sql
  3  WHERE sql_id = 'f364ymn1bbr4q'
  4  ORDER BY child_number;

CHILD_NUMBER IS_BIND_SENSITIVE IS_BIND_AWARE IS_SHAREABLE PLAN_HASH_VALUE
------------ ----------------- ------------- ------------ ---------------
           0 Y                 Y             Y                 4270555908
           1 Y                 Y             Y                 2966233522
```

概括起来，为了增加查询优化器产生高效执行计划的可能性，就不应该使用绑定变量。绑定变量扫视可能会有帮助。然而，有时候能否产生高效的执行计划只是运气的问题。唯一的例外是从11.1版本开始，新的自适应游标共享能自动识别出问题。

3. 最佳实践

任何特性都应该仅在使用它的收益比损害要大时才使用。某些情况下很容易做决定。例如，当执行一个没有WHERE条件的SQL语句（例如简单的INSERT语句）时没有理由不使用绑定变量。另一方面，当被绑定变量扫视破坏的风险较高时，无论如何也不应该使用绑定变量。尤其是遇到下面三种情况时。

- 当查询优化器必须检查一个值是否在可访问值的范围之外时（也就是比列中最小值小或比最大值大）。
- 当WHERE条件中的谓词是基于范围条件的（例如，HIREDATE >'2009-12-31'）。
- 当查询优化器使用直方图时。

因此，对于可共享的游标，当遇到以上三种情况时就不应该使用绑定变量。对于其他所有的情况，就没有这么绝对了。然而，最好考虑以下两种主要的情况。

- **SQL语句处理少量数据**：每当处理较少的数据时，硬解析的时间可能会接近或者超过执行时间。在这种情况下，使用绑定变量从而避免硬解析通常是必须的。在SQL语句预计会经常执行时尤其如此。通常这样的SQL语句用于数据实体系统（一般是OLTP系统相关的）。
- **SQL语句处理大量数据**：每当处理大量数据时，硬解析时间通常比执行时间小几个量级。在这种情形下，使用绑定变量不仅对于整个响应时间是无关紧要的，同时也增加了查询优化器产生非常低效的执行计划的风险。因此，不应该使用绑定变量。通常，这样的语句用来做批处理任务，用于报表用途，或者在数据仓库环境下由OLAP应用和BI工具发出。

2.5 读写数据块

为了读写数据文件中的数据块，数据库引擎利用几种不同的磁盘I/O操作（参见图2-4）。

- ❑ 逻辑读：服务进程在访问一个缓冲区缓存中的块或进程私有内存中的块时执行逻辑读。注意，逻辑读既用于读，同时也用于向一个数据块写数据。
- ❑ 缓冲区缓存读：当服务进程需要的块还不在缓冲区缓存时执行缓冲区缓存读。所以它会打开数据文件，读这个数据块，然后将其存储在缓冲区缓存中。
- ❑ DBWR写：通常情况下，服务进程不会向数据文件写数据，它们只修改存储在缓冲区缓存中的块。然后由数据库写进程（即后台进程）负责将修改的块（也称为脏块）存储到数据文件中。
- ❑ 直接路径读：在某些情况下（在第13章和第15章中详述），服务进程能够直接从数据文件中读取数据块。当服务进程使用这种方式时，数据块会直接传输至进程的私有内存而非加载至缓冲区缓存内。
- ❑ 直接路径写：在某些情况下（在第15章中详述），服务进程能够直接向数据文件写入数据块。

在不必区分那些是否涉及缓冲区缓存的磁盘I/O操作的情况下，可以使用以下两个术语。

- ❑ 物理读包含缓冲区缓存读和直接路径读。
- ❑ 物理写包含直接路径写和DBWR写。

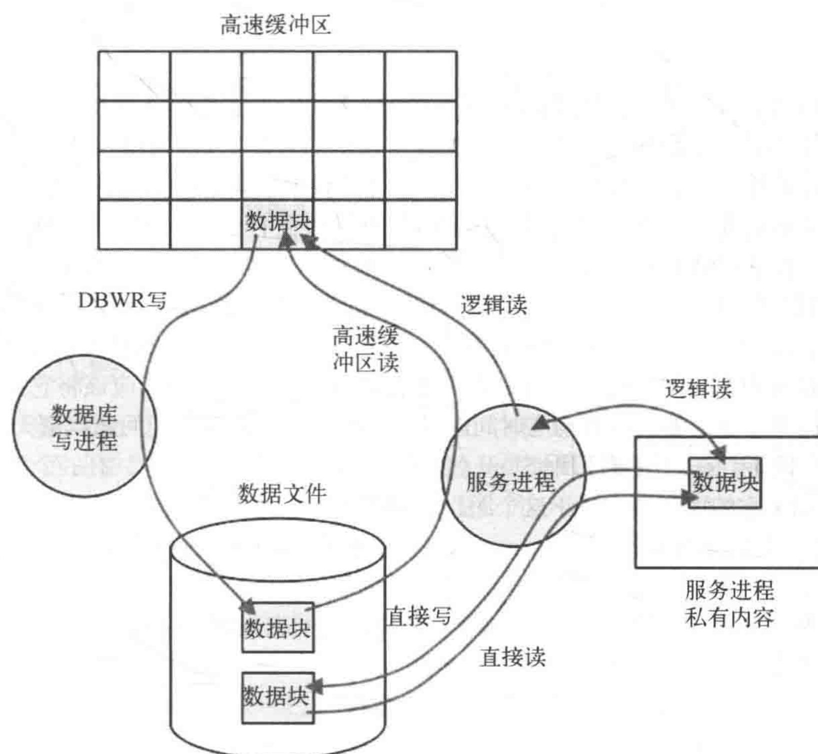


图2-4　数据库引擎使用几种不同类型的I/O操作

当数据文件存储在一台Exadata存储服务器上时，数据库引擎也可以利用第六种磁盘I/O操作：智能扫描（smart sacn）。简言之，在Exadata系统上，数据库引擎可以使用智能扫描替代直接路径读。从数据库引擎的角度来看，这两种磁盘I/O操作的显著区别是直接路径读返回规则的块，而智能扫描返回不同的数据结构。智能扫描的目标是降低一部分原本应由数据库引擎向存储层完成的工作。考虑到这一点，使用智能扫描有下面三个主要的目标。

- 避免在Exadata存储服务器和数据库实例中移动不相干的数据。
- 允许Exadata存储服务器避免读取不需要的磁盘数据。
- 减少Exadata存储服务器的CPU密集型操作，进而减少数据库引擎的CPU使用率。

什么是Oracle Exadata

Exadata是由Oracle设计的数据库一体机，由数据库服务器、Exadata存储服务器、拥有InfiniBand技术的光纤存储网络以及其他所有运行Oracle数据库所需的组件组成。然而，Exadata并不仅仅是一套硬件解决方案。当Oracle数据库在Exadata硬件上运行时，设计用于获取更好性能的特别软件特性就启用了。其中的一个关键设计决策就是，将部分本应由数据库服务器完成的工作负载转移给存储服务器。

2.6　检测

正如第1章提到的，每个应用程序都应该进行检测。换句话说，问题不是应不应该去做，而是应该怎么做。这是架构师在新的应用程序开发之初就应做出的重要决定。虽然检测代码通常是为了在异常条件下具体化应用程序的行为而实现的，但是它也可以用来调查性能问题。为了定位性能问题，我们尤其想了解执行过哪些操作、执行顺序如何、处理的数据有多少、这些操作执行了多少次以及花费了多长时间。在某些情形下（例如大型任务），了解使用了多少资源也是有帮助的。由于在调用级别或者链路级别的信息已由代码探查器提供，检测时应该特别关注业务相关的操作以及各个组件（层）之间的交互。此外，如果一个请求需要在同一个组件内部执行复杂的处理，可以提供处理过程中实施的主要步骤的相应时间。换句话说，为了更有效地利用检测代码，应该将它添加到代码的决策性位置上。我强调一下，如果没有响应时间的信息，检测对于调查性能问题就毫无意义。

我们来看一个例子。在应用程序JPetStore（在第1章中简单介绍过）中，有一个叫作Sign-on登入的操作，图2-5展示了它的序列图。基于这个图，检测至少应该提供如下这些信息。

- 从servlet[①]给出响应到请求（`FrameworkServlet`）来衡量的请求的系统响应时间。这是业务相关的操作。
- SQL语句和数据访问对象（`AccountDao`）与数据库之间交互的响应时间。这是中间层和数据库层之间的交互。
- 请求以及与数据库的交互两者开始和结束的时间戳。

① servlet是一种向来自Web客户端的请求做出响应的Java程序，运行于J2EE应用服务器中。

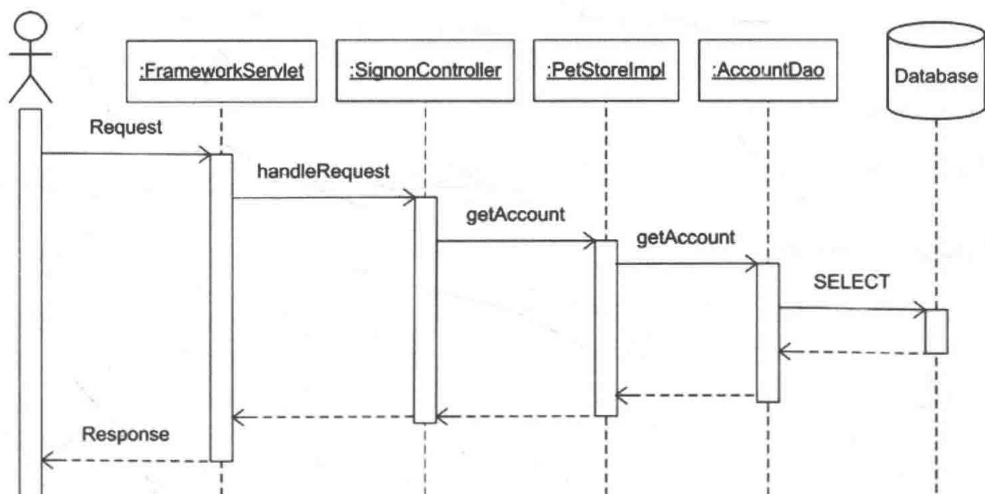


图2-5 JPetStore登入操作的序列图

通过这些值和用户响应时间，如果可以再利用应用程序，就可以轻易使用手表检测它们，从而就可以通过类似图1-7的方式拆分响应时间。

在实践中，不能随意在你想要的位置添加检测代码。在图2-5的案例中，你有两个问题。一是servlet（FrameworkServlet）是由Spring框架提供的class文件。因此，你不想去修改它。二是数据访问对象（AccountDao）只不过是在持久层框架（本例中是iBatis）之前使用的一个接口。因此，你也不能向它加入代码。对于第一个问题，可以通过加入检测代码，从而创建自己的servlet继承FrameworkServlet来解决。对于第二个问题，为了方便起见，可以决定只检测持久层框架的调用。这应该没问题，因为数据库本身已经经过检测了，所以，如果有必要，就能够判定持久层框架本身的消耗。

现在已经看到如何决定应该将检测代码加到哪里，我们可以看一下关于如何在应用程序代码中落实它的具体例子。随后，我们将查看Oracle特有的数据库调用的检测。

2.6.1 应用程序代码

通常，检测代码都是通过利用已经可用的日志框架来实现的。原因很简单：写一个快速而灵活的日志框架并不简单。因此使用已有的框架可以节省大量的开发时间。事实上，使用日志的主要缺点是，不恰当的实现会拖慢应用程序。为了避免这个问题，开发人员不仅应该限制日志的冗长，而且需要用一个高效的日志框架实现它。

Apache日志服务项目（Apache Logging Services Project[①]）为日志框架树立了良好的范例。这个项目的核心是log4j，它是一个为Java写的日志框架。因为在Java上的成功，它也被移植到其他的编程语言中，如C++、.NET、Perl及PHP等。在这里我会提供一个基于log4j和Java的例子。举例来说，如果你想检测图2-5中SignonController servlet的handleRequest方法的响应时间，可以编写如下的代码：

① 查看http://logging.apache.org获取更多信息。

```
public ModelAndView handleRequest(HttpServletRequest request,
                                  HttpServletResponse response) throws Exception
{
  if (logger == null)
  {
    logger = Log4jLoggingHelper.getLog4jServerLogger();
  }

  if (logger.isInfoEnabled())
  {
    long beginTimeMillis = System.currentTimeMillis();
  }

ModelAndView ret = null;
String username = request.getParameter("username");

// 处理请求的代码……

  if (logger.isInfoEnabled())
  {
    long endTimeMillis = System.currentTimeMillis();
    logger.info("Signon(" + username + ") response time " +
            (endTimeMillis-beginTimeMillis) + " ms");
  }

  return ret;
}
```

简单来说，检测代码，也就是粗体部分，会在方法开始时获取一个时间戳，在方法结束时也获取一个时间戳，然后记录一条信息，其中包含用户的名称以及执行方法花费的时间。开始的时候，它还会检查日志记录器是否已经被初始化了。注意，`Log4jLoggingHelper`类针对Oracle WebLogic Server，而非log4j。这是测试时使用的。虽然这段代码简单明了，但还是有下面几点需要我们注意。

- 开始检测时间，本例中是通过在检测代码的最开始调用`currentTimeMillis`方法。你永远不知道会遇到什么情况，甚至初始化代码也会消耗时间。
- 日志框架应该提供不同级别的消息。在log4j中有以下级别可用（按冗长度排序）：`fatal`、`error`、`warning`、`information`和debug。通过调用下列方法之一来设置级别：`fatal`、`error`、`warn`、`info`和debug。换言之，每个级别都有它对应的方法。通过将消息放入不同的级别，你可以通过启用和禁用指定的级别明确选择日志的冗长度。如果你想在调查某个问题时仅启用部分检测代码，这将会非常有用。
- 即使日志程序知道启用了哪个日志级别，也最好不要调用日志方法。以`info`为例，如果那个指定级别的日志没有启用，就尤其要避免消息构建的开销（以及随之而来的调用垃圾回收）。通常会调用像`isInfoEnabled`这样的方法来检查指定级别的日志是否打开，如果有必要再调用日志方法，这样要快得多。这样做可以导致巨大的差别。举例来说，在我的测试服务器上，调用`isInfoEnabled`大概花费7纳秒，而调用`info`方法并提供之前代码段中的参数花费大约265纳秒（我使用`LoggingPerf.java`中定义的类来检测这些统计信息）。另一种减小检测开销的技术是在正式编译时移除日志代码。但这不是一个首选的技术，因为通常情况下在需要检测时是

不可能动态重新编译该代码的。进一步讲，正如你所看到的，一行不产生任何消息的检测代码的开销真的很小。

- 在这个例子中，产生的消息可能是类似Signon(JPS1907) response time 24 ms这样的内容。这对人类是友好的，但如果你计划将消息传递给另一个程序用于检查服务等级协议，像XML或者JSON这样的更结构化的形式可能会更合适一些。

2.6.2 数据库调用

本节将讨论如何正确检测数据库调用，以便为数据库引擎提供关于它们将要执行的应用程序上下文的信息。要意识到这种类型的检测与你在上一节中看到的有很大不同。事实上，数据库调用不仅应该能够像应用程序的任何其他部分一样进行检测，而且数据库自身也能够生成关于它执行的数据库调用的详细信息。你会在本书第二部分了解到更多内容。这里描述的这种类型的检测目标是，向数据库引擎提供关于使用它的用户或者应用程序的信息。这样做是有必要的，因为数据库引擎通常只有少量关于应用程序代码、会话以及最终用户之间的关系的信息，甚至根本没有，考虑如下两个常见的情形。

- 数据库引擎并不知道应用程序代码的哪个部分正在通过会话执行SQL语句。例如，数据库引擎对于究竟是哪个模块、类或者报表正通过给定的会话执行一个特定的SQL语句毫无线索。
- 如果连接池是由一名技术用户打开的，而代理用户没有使用它，那么当应用程序通过该连接池连接到数据库引擎时，最终用户的身份验证通常都是由应用程序自己执行的。因此，数据库引擎会忽略是哪个最终用户在使用哪个会话。

基于这些原因，数据库引擎提供了动态关联一个数据库会话的如下特性的机会。

- 客户端标识符（Client identifier）：用于识别客户端的64字节长度的字符串，尽管不是非常明确。
- 客户端信息（Client information）：用于描述客户端的64字节长度的字符串。
- 模块名称（Module name）：用于描述会话中正在使用的模块名称的48字节长度的字符串。
- 动作名称（Action name）：用于描述正在处理的动作的32字节长度的字符串。

警告 对于通过数据库链接打开的会话，只有客户端标识符会自动传播到远端的会话上。因此对于其他特性，有必要显式地设定。

客户端特性的值通过v$session视图和userenv上下文环境体现。下面的例子是对session_attributes.sql脚本生成的输出的一段摘录，展示了如何查询它们。

```
SQL> SELECT sys_context('userenv','client_identifier') AS client_identifier,
  2         sys_context('userenv','client_info') AS client_info,
  3         sys_context('userenv','module') AS module_name,
  4         sys_context('userenv','action') AS action_name
  5  FROM dual;

CLIENT_IDENTIFIER    CLIENT_INFO  MODULE_NAME      ACTION_NAME
-------------------- ------------ ---------------- ------------------------
helicon.antognini.ch Linux x86_64 session_info.sql test session information

SQL> SELECT client_identifier,
```

```
  2         client_info,
  3         module AS module_name,
  4         action AS action_name
  5  FROM v$session
  6  WHERE sid = sys_context('userenv','sid');

CLIENT_IDENTIFIER    CLIENT_INFO  MODULE_NAME      ACTION_NAME
-------------------- ------------ ---------------- ------------------------
helicon.antognini.ch Linux x86_64 session_info.sql test session information
```

注意，其他显示SQL语句的视图也包含module和action列，例如v$sql。提醒一句：这些特性关联到具体的会话，但是一个给定的SQL语句可以被多个会话共享从而拥有不同的模块名称和动作名称。这些由动态性能视图显示的值是第一个解析SQL语句的会话在做硬解析时设置的。如果你不加小心，可能会误入歧途。

现在你已经看到了什么是可用的，我们来看一下如何设置这些值。第一个方法——PL/SQL，是唯一一个不需要依赖连接数据库的接口的方法，因此它可以用于大多数情形中。接下来的四个——OCI、JDBC、ODP.NET和PHP，则仅可以通过指定的接口程序使用。它们的主要优势是这些值会加入到下一次数据库调用中而不会产生额外的往返操作，而调用PL/SQL却会。因此为它们设置这些特性的负载可以忽略不计。

1. PL/SQL

你需要使用dbms_session包中的存储过程set_identifier来设置客户端标识符。在某些情况下，比如客户端标识符是在全局上下文环境以及连接池中使用时，可能有必要清除已与给定会话关联的值。如果是这样，就可以用clear_identifier这个存储过程。

要设置客户端信息、模块名称以及动作名称，可以使用dbms_application_info包中对应的set_client_info、set_module和set_action存储过程。为简单起见，set_module存储过程不仅接受模块名称，而且也接受动作名称。

下面的PL/SQL代码块摘自脚本session_attributes.sql，它展示了这样一个例子。

```
BEGIN
  dbms_session.set_identifier(client_id=>'helicon.antognini.ch');
  dbms_application_info.set_client_info(client_info=>'Linux x86_64');
  dbms_application_info.set_module(module_name=>'session_info.sql',
                                   action_name=>'test session information');
END;
```

2. OCI

为了设置这四个特性，你可以使用OCIAttrSet函数。第三个参数指定特性的值。第五个参数借助于下列常量中的一个来指定要设置哪个特性。

- OCI_ATTR_CLIENT_IDENTIFIER
- OCI_ATTR_CLIENT_INFO
- OCI_ATTR_MODULE
- OCI_ATTR_ACTION

下面的代码片段，是session_attributes.c文件中的一段摘录，展示了如何调用OCIAttrSet函数来

设置客户端标识符。

```
text client_id[64] = "helicon.antognini.ch";
OCIAttrSet(ses,                           // 会话句柄
           OCI_HTYPE_SESSION,             // 被修改的句柄类型
           client_id,                     // 特性的值
           strlen(client_id),             // 特性值的大小
           OCI_ATTR_CLIENT_IDENTIFIER,    // 要设置的特性
           err);                          // 错误句柄
```

3. JDBC

JDBC有两种方式设置会话的特性。传统方式是使用Oracle扩展功能，而现代方式是基于标准的JDBC应用编程接口。现代方式在Oracle提供的12.1版的JDBC驱动中可用。因为这种方式是基于和传统方式相同的协议，也可以和10.2、11.1及11.2版本的数据库一起使用。注意，从12.1版本开始，传统方式被弃用了。

- **传统方式**

为了设置客户端标识符、模块名称和动作名称，可以用由OracleConnection接口提供的setEndToEndMetrics方法。没有提供对客户端信息设置的支持。通过字符串数组向方法传递一个或多个特性。数组中的位置由以下常量定义，用于确定设置哪个特性：

- END_TO_END_CLIENTID_INDEX
- END_TO_END_MODULE_INDEX
- END_TO_END_ACTION_INDEX

下面的代码片段是SessionAttributes.java文件的一段摘录，展示了如何定义包含特性的数组以及如何调用setEndToEndMetrics方法：

```
metrics = new String[OracleConnection.END_TO_END_STATE_INDEX_MAX];
metrics[OracleConnection.END_TO_END_CLIENTID_INDEX] = "helicon.cha.trivadis.com";
metrics[OracleConnection.END_TO_END_MODULE_INDEX] = "SessionAttributes.java";
metrics[OracleConnection.END_TO_END_ACTION_INDEX] = "test session information";
((OracleConnection)connection).setEndToEndMetrics(metrics, (short)0);
```

- **现代方式**

为设置客户端标识符、模块名称以及动作名称，你可以使用Connection接口提供的setClientInfo方法。不提供设置客户端信息的支持。setClientInfo方法接受两个字符串参数：特性的名称和值。你必须按如下值指定特性名称：

- OCSID.CLIENTID
- OCSID.MODULE
- OCSID.ACTION

下面的代码片段来自于SessionAttributes12c.java文件中的一段摘录，展示了如何通过setClientInfo设置特性：

```
connection.setClientInfo("OCSID.CLIENTID", "helicon.cha.trivadis.com");
connection.setClientInfo("OCSID.MODULE", "SessionAttributes12c.java");
connection.setClientInfo("OCSID.ACTION", "test session information");
```

4. ODP.NET

要设置这四个特性，可以使用OracleConnection类的ClientId、ClientInfo、ModuleName和ActionName等属性。注意除了ClientId属性，其余的仅从11.1.0.6.20版本起才可以使用。为了防止堆积的会话接手这些设置，属性的值会在调用OracleConnection类的Close或Dispose方法时设置为null。下面的代码片段是来自SessionAttributes.cs文件的摘录，展示了如何设置这些属性：

```
connection.ClientId = "helicon.antognini.ch";
connection.ClientInfo = "Linux x86_64";
connection ModuleName = "SessionAttributes.cs";
connection.ActionName = "test session information";
```

5. PHP

要设置这四个特性，请使用PECL OCI8扩展中提供的函数。每个函数接受两个参数（连接和特性值）作为输入，并且返回一个Boolean型值（成功时为TRUE，失败时为FALSE）。这四个函数如下所示：

- oci_set_client_identifier
- oci_set_client_info
- oci_set_module_name
- oci_set_action

下面的代码片段是来自session_attributes.php脚本的一段摘录，展示了如何设置所有的特性：

```
oci_set_client_identifier($connection, "helicon.antognini.ch");
oci_set_client_info($connection, "Linux x86_64");
oci_set_module_name($connection, "session_attributes.php");
oci_set_action($connection, "test session information");
```

注意，这些函数仅在同时符合以下两个条件时才可用：OCI8的版本为1.4及以上，OCI8与10.1及以上版本的客户端库相关联。

2.7 小结

本章描述了当数据库引擎解析和执行SQL语句时实施的操作，着重讨论了与使用绑定变量有关的正反两方面的理由。此外，还介绍了常用的术语并描述了如何检测应用程序代码和数据库调用。

本书第二部分将致力于回答图1-4抛出的前两个问题：

- 时间是在哪里消耗的
- 时间是如何消耗的

简单来说，第二部分中的三章介绍了查明问题所在以及查明导致问题的原因的相关方法。因为你在修复性能问题上没有退路了，所以必须正确回答这些问题。如果你不知道引起问题的原因，自然就不可能修复这个问题。

Part 2 第二部分

识　别

让我们把问题解决掉，不要让猜测使它变得复杂。[①]

——Eugene F. Kranz

当程序出现性能问题时，很显然要做的第一件事就是识别导致问题的根源。不幸的是，这往往是麻烦的开始。在一个典型的案例中，当所有人都在寻找性能问题的根源时，开发人员开始抱怨数据库性能差，而数据库管理员则一方面抱怨开发人员滥用数据库，另一方面又抱怨存储子系统管理员，理由是昂贵的硬件并没有带来更好的性能。并且随着应用的复杂度和基础设施支持的增加，这种互相埋怨的混乱局面更是一发不可收拾。

第二部分的章节旨在介绍数据库的一些关键特性，利用它们可以找出在数据库内部时间消耗在了哪里以及消耗的方式。在阅读这些章节时，请记住第 1 章的建议，仅当端到端的响应时间数据表明数据库层面可能存在问题时，才收集数据库引擎执行时的详细信息。否则，你可能会进行错误的分析。如果你分析错误，那么在诊断故障时，就很可能无法判断出导致性能问题的原因。首先要确定问题是出现在数据库层面，然后利用这部分的内容来具体操作。

下一步要考虑的是，当处理性能问题时，是否能任意地重现问题。如果可以，那么事情会变得很简单。如果能重现问题，那么应该很容易就能找出问题所在，这在第 3 章会讲到。如果问题不能随意重现，那么就会有两个选择：等待问题再次出现，或者查找历史性能指标的知识库。这两种方式会分别在第 4 章和第 5 章中介绍。

此外，无论能否重现问题，都必须找出最耗时的 SQL 或者 PL/SQL 代码调用。然后，针对每条耗时的语句，应该收集任何可以帮助诊断问题的附加信息。

这些附加信息通常包括：执行计划、关键运行时统计（比如已处理的行数和 CPU 使用率的数量）以及已经历的等待事件。第二部分只讲了如何收集这些信息，并未介绍如何处理这些信息。第三部分和第四部分会对如何处理这些信息详细讲解。

---

① 引自朗·霍华德执导的电影《阿波罗 13 号》。这句话就出现在那句著名的“休斯顿，我们遇到了麻烦”（Houston, we have a problem）之后大约 3 分钟处。

当你的分析指向几条 SQL 语句或几行 PL/SQL 代码，会发现部分程序需要优化，并且可以很简单地解决它。否则，大量 SQL 语句或者庞大的 PL/SQL 代码导致的过长响应时间，往往意味着存在设计问题。这时彻底重新设计有时是必要的。如果设计没有问题，那么很可能是运行程序的服务器性能不足造成的。

第二部分的最终目标是针对性能差的程序介绍并使用一种找到瓶颈的方法，而不是仅仅靠猜测。这个目标还包括收集在第三部分和第四部分需要处理的基本信息。

第3章 分析可重现的问题

当一个程序出现问题，想要重现问题最有效的办法就是利用有效跟踪和剖析特性来定位问题。首先，把问题分为以下三类。

- 数据库引擎将大量时间花费在执行SQL语句上。
- 数据库引擎将大量时间花费在执行PL/SQL代码上。
- 数据库引擎（几乎）空闲。换句话说，瓶颈不在数据库层。

要将问题分类，首先要从跟踪数据库调用开始。如果分析指向SQL语句，那么最初对问题进行分类所使用的跟踪文件已经包含了所有必要的信息。如果问题出在PL/SQL代码，则应该剖析PL/SQL代码。否则，问题就不在数据库层，那么分析应该继续剖析数据库引擎执行之外的应用代码。

本章目标不仅仅是介绍Oracle数据库提供的跟踪和剖析功能，而且还会给出你分析问题时所使用的工具的示例。最后，本章会展示这些工具如何帮助你快速、高效地定位性能问题。

3.1 跟踪数据库调用

当性能瓶颈出现在数据库层时，就更应该关注应用与数据库引擎的交互。Oracle数据库是一款高度工具化的软件，多亏有了SQL跟踪（SQL trace），它提供的跟踪文件不仅包含执行的SQL语句列表，同时也包含了这些语句处理时的深度性能指标。

图3-1展示了涉及跟踪数据库调用时必不可少的阶段。后续几节，伴随对SQL跟踪的解释，会详细讨论每一个阶段。

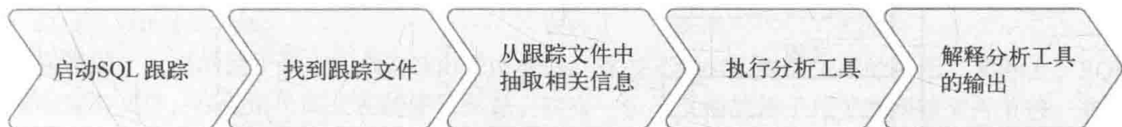


图3-1 跟踪数据库调用时必不可少的阶段

3.1.1 SQL 跟踪

第2章提到过，要处理SQL语句，数据库引擎（尤其是SQL引擎）会执行数据库调用（例如解析、执行和获取）。图3-2总结了对于每个数据库调用，SQL引擎可以执行以下操作：

- 使用CPU自己处理这些调用；

- 使用其他资源（比如磁盘）；或者
- 不得不通过一个同步点来保证数据库引擎的多用户处理能力（比如闩）。

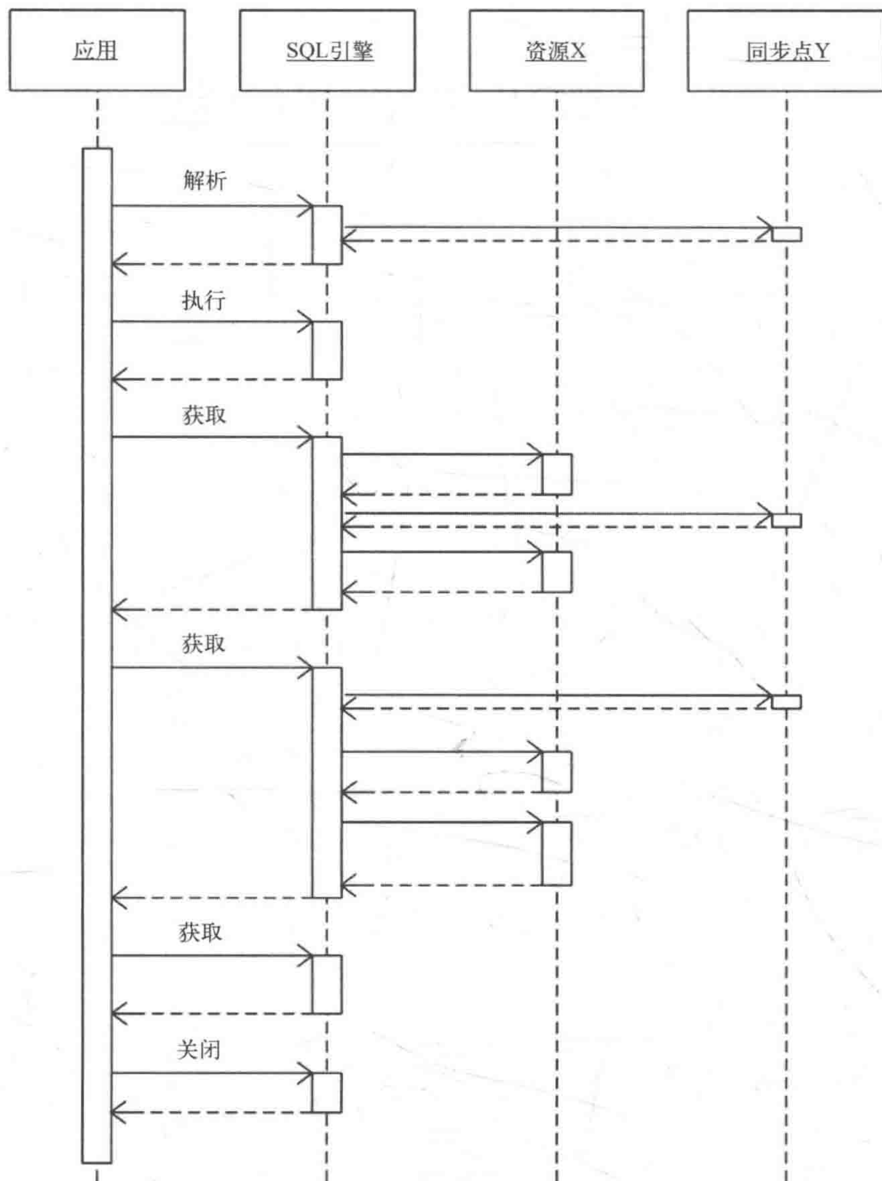


图3-2　SQL引擎与其他组件间交互的顺序图

SQL跟踪的目的有两个：首先，为分解服务时间与等待时间之间的响应时间提供信息；其次，为使用的资源和同步点提供详细信息。所有关于SQL引擎与其他组件间的交互信息都会保存在跟踪文件里。请注意，在图3-2中，CPU、资源X和同步点Y的属性都是人工的。其原因是为了展示每个调用可能会以不同的方式使用数据库引擎。

尽管本章后续部分还会介绍更多的细节，但让我们暂时先看一个由SQL跟踪提供的信息，该信息可以使用工具抽取出来（这里使用TKPROF）。它包含了SQL语句的文本、一些执行统计、处理SQL时发生的等待事件，以及解析阶段的信息，例如生成执行计划。请注意，这些信息是由程序执行的每条SQL语句和数据库引擎自身递归调用产生的。

```
SELECT CUST_ID, EXTRACT(YEAR FROM TIME_ID), SUM(AMOUNT_SOLD)
FROM SALES
WHERE CHANNEL_ID = :B1
GROUP BY CUST_ID, EXTRACT(YEAR FROM TIME_ID)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch      164      0.84       1.27       3472       1781          0       16348
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      166      0.84       1.28       3472       1781          0       16348

Misses in library cache during parse: 1
Misses in library cache during execute: 1
Optimizer mode: ALL_ROWS
Parsing user id: 77 (SH)   (recursive depth: 1)
Number of plan statistics captured: 1

Rows (1st) Rows (avg) Rows (max)  Row Source Operation
---------- ---------- ----------  ---------------------------------------------------
     16348      16348      16348  HASH GROUP BY
    540328     540328     540328   PARTITION RANGE ALL PARTITION: 1 28
    540328     540328     540328    TABLE ACCESS FULL SALES PARTITION: 1 28

Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  Disk file operations I/O                        2        0.00          0.00
  db file sequential read                        29        0.00          0.00
  direct path read                               70        0.00          0.00
  asynch descriptor resize                       16        0.00          0.00
  direct path write temp                       1699        0.02          0.62
  direct path read temp                        1699        0.00          0.00
```

之前提到，上面的例子是由工具TKPROF生成的。这并不是SQL跟踪生成的输出。实际上，SQL跟踪输出文本文件，存储组件间交互的原始信息。这有一份与上面的例子相关的跟踪文件节选。通常情况下，对应每一个调用或者等待，跟踪文件中至少会存在一行代码。

```
...
...
PARSING IN CURSOR #140105537106328 len=139 dep=1 uid=77 oct=3 lid=93 tim=1344867866442114
hv=2959931450 ad='706df490' sqlid='arc3zqqs6ty1u'
SELECT CUST_ID, EXTRACT(YEAR FROM TIME_ID), SUM(AMOUNT_SOLD) FROM SALES WHERE CHANNEL_ID = :B1
GROUP BY CUST_ID, EXTRACT(YEAR FROM TIME_ID)
END OF STMT
PARSE #140105537106328:c=1999,e=1397,p=0,cr=0,cu=0,mis=1,r=0,dep=1,og=1,plh=0, tim=1344867866442113
BINDS #140105537106328:
```

```
  Bind#0
   oacdty=02 mxl=22(21) mxlc=00 mal=00 scl=00 pre=00
   oacflg=03 fl2=1206001 frm=00 csi=00 siz=24 off=0
   kxsbbbfp=7f6cdcc6c6e0 bln=22 avl=02 flg=05
   value=3
EXEC #140105537106328:c=7000,e=7226,p=0,cr=0,cu=0,mis=1,r=0,dep=1,og=1,plh=3604305554,
tim=1344867866449493
WAIT #140105537106328: nam='Disk file operations I/O' ela= 45 FileOperation=2 fileno=4 filetype=2
obj#=69232 tim=1344867866450319
WAIT #140105537106328: nam='db file sequential read' ela= 59 file#=4 block#=5009 blocks=1 obj#=69232
tim=1344867866450423
...
...
FETCH #140105537106328:c=0,e=116,p=0,cr=0,cu=0,mis=0,r=48,dep=1,og=1,plh=3604305554,
tim=1344867867730523
STAT #140105537106328 id=1 cnt=16348 pid=0 pos=1 obj=0 op='HASH GROUP BY (cr=1781 pr=3472 pw=1699
time=1206229 us cost=9220 size=4823931 card=229711)'
STAT #140105537106328 id=2 cnt=540328 pid=1 pos=1 obj=0 op='PARTITION RANGE ALL PARTITION: 1 28
(cr=1781 pr=1773 pw=0 time=340163 us cost=1414 size=4823931 card=229711)'
STAT #140105537106328 id=3 cnt=540328 pid=2 pos=1 obj=69227 op='TABLE ACCESS FULL SALES PARTITION: 1
28 (cr=1781 pr=1773 pw=0 time=280407 us cost=1414 size=4823931 card=229711)'
CLOSE #140105537106328:c=0,e=1,dep=1,type=3,tim=1344867867730655
...
...
```

在上面节选的部分里，一些描述此类信息的标记以粗体突出显示出来。

- PARSING IN CURSOR和END OF STMT之间的部分就是SQL语句的文本。
- PARSE、EXEC、FETCH和CLOSE分别表示解析、执行、获取和结束调用。
- BINDS表示绑定变量的定义和值。
- WAIT表示处理过程中发生的等待事件。
- STAT表示已发生的执行计划和关联的统计信息。

你可以在Oracle Support文档*Interpreting Raw SQL_TRACE output*（39817.1）中找到关于跟踪文件格式的简单描述。如果对这个话题感兴趣，想了解详细描述和相关讨论，可以阅读Millsap的著作*The Method R Guide to Mastering Oracle Trace Data*（CreateSpace，2013）。

在数据库内部，SQL跟踪基于调试事件10046。表3-1描述了可支持的级别，这代表在跟踪文件里可获得的信息量。将SQL跟踪设置为高于等级1时，SQL跟踪也被称为扩展SQL跟踪。

表3-1　10046调试事件级别

级　别	描　述
0	调试事件被禁止
1	调试事件启用。针对每个处理的数据库调用，都会给出以下信息：SQL语句、响应时间、服务时间、处理的行数、逻辑读数、物理读数和物理写数、执行计划以及一小部分附加信息 在10.2版本中，执行计划只有在与其关联的游标被关闭后才会写入到跟踪文件中。与执行计划关联的统计信息，来自多次执行的值聚合 从版本11.1起，执行计划在每个游标第一次执行时写入跟踪文件。与执行计划关联的统计信息，仅仅来自第一次执行的值
4	同级别1，附加信息是绑定变量。主要是数据类型及其精度和针对每次执行使用的值

（续）

级 别	描 述
8	同级别1，附加等待时间的详细信息。对于每次处理时经历的等待事件，会给出一些信息，包括等待事件名称、持续时间和一些附加参数，用来确认被等待的资源
16	同级别1，每次执行后的执行计划信息也会写入跟踪文件。仅对版本11.1及以上版本有效
32	同级别1，但不包含执行计划信息。仅对版本11.1及以上版本有效
64	同级别1，第一次执行之后的执行计划信息也可能会被写入。条件是最后一次写入执行计划信息后，某个游标还会至少占用一分钟的数据库时间。这个级别适用于两种情况：(1) 当第一个执行的信息不足以分析某些特殊情况时；(2) 当写入每个执行的信息开销过大时（如级别16）。仅对版本11.2.0.2[①]及以上版本有效

除了表3-1描述的级别外，你也可以把级别4和级别8与大于级别1的其他级别相加。比如以下几种情形。

级别 12(4+8)：同时应用级别4和级别8。

级别 28(4+8+16)：同时应用级别4、级别8和级别16。

级别 68(4+64)：同时应用级别4和级别64。

下一部分会介绍如何启用和禁用SQL跟踪，如何配置环境以对我们有利，以及如何找到生成的跟踪文件。

调试事件

调试事件是由数字来识别的，是用来在一个运行的数据库引擎进程中设置一种标志的方法。目的是改变它的行为，例如，启用或者禁止一个特性，测试或模拟一个讹误或事故，收集跟踪或者调试信息。一些调试事件不是简单的标志，可以在N个级别中启用。每个级别都有自己的动作。在一些情况下，级别是一个块或者内存结构的地址。

应该仅在Oracle Support指导下或者在知道调试事件会导致哪些改变的情况下，小心使用调试事件。调试事件启用的是特定码路径。因此，当调试事件引起问题时，应该在不使用调试事件的情况下确认问题是否能重现。

几乎没有调试事件会被Oracle记录在文档里。如果有文档的话，通常可以在Oracle Support文档中找到。换句话说，调试事件通常不会记录在数据库引擎的Oracle官方文档里。你可以在$ORACLE_HOME/rdbms/mesg/oraus.msg文件中找到完整的可用调试事件列表。请注意，此文件不是存在于所有平台的版本中。10 000到10 999被留作调试事件。

1. 使用ALTER SESSION启用SQL跟踪

*SQL Language Reference*手册中记录着ALTER SESSION语句，可用来启用SQL跟踪。请看下例：

```
ALTER SESSION SET sql_trace = TRUE
```

你仅可以使用ALTER SESSION语句将sql_trace设置为TRUE，这相当于级别1。在实际工作中，级别1通常是不够的。在大多数情况下，你需要把响应时间彻底拆开，以弄清楚瓶颈到底在哪里。基于这个

① 或者安装包含修复bug 8328200的补丁包（比如11.2.0.1.0 Bundle Patch 7 for Exadata）。

原因，我不会再过多介绍这种启用SQL跟踪的方法。我要介绍的是Oracle Support文档*EVENT: 10046 "enable SQL statement tracing (including binds/waits)"*（21154.1）中介绍的可以启用任何级别SQL跟踪的方法。要启用和禁用任意级别的SQL跟踪，需要执行ALTER SESSION语句来设置事件的初始化参数。下面的SQL是在当前会话启动级别12的SQL跟踪，请注意事件编号和级别的写法。

```
ALTER SESSION SET events '10046 trace name context forever, level 12'
```

接下来的SQL会禁用SQL跟踪，请注意这里不是通过指定为级别0来禁用。

```
ALTER SESSION SET events '10046 trace name context off'
```

你也可以使用ALTER SYSTEM语句来设置事件初始化参数。该语句的语法和ALTER SESSION是一样的。任何情况下，在系统级别设置SQL跟踪都是没有意义的，此外这么做还会造成庞大的开销。请注意，这只对启用SQL跟踪后的会话有效。

2. 使用DBMS_MONITOR启用SQL跟踪

Oracle数据库也提供了dbms_monitor包来启用和禁用SQL跟踪。这个包不仅提供了一种启用会话级别的扩展SQL跟踪方法，更重要的是，你可以基于会话属性来启用和禁用SQL跟踪（参见第2章）。这些属性包括：客户端标识符、服务名、模块名和动作名。这意味着如果应用配置正确，你可以针对执行数据库调用的会话单独启用和禁用SQL跟踪。目前，这是特别有用的方法，因为在大多数情况下都会用到连接池，所以用户已经不会关联某个特定的会话。

当使用dbms_monitor包时，不需要直接指定诊断事件10046的级别。每个过程提供三个参数（binds、waits以及自版本11.1起才有的plan_stat）来启用SQL跟踪。使用以下参数可以启用对应的级别。

- 启用级别4，binds需要设置为TRUE。
- 启用级别8，waits需要设置为TRUE。
- 启用级别16，plan_stat需要设置为all_executions。
- 启用级别32，plan_stat需要设置为never。
- dbms_monitor无法启用级别64。

参数waits的默认值为TRUE。binds的默认值为FALSE。plan_stat的默认值为NULL（相当于first_execution）。因此，默认级别是8。

接下来的内容给出了一些使用dbms_monitor包在会话、客户端、组件和数据库级别启用和禁用SQL跟踪的例子。请注意，在默认情况下，只有拥有dba角色的用户可以执行dbms_monitor包下的过程。

● 会话级别

为会话启用和禁用SQL跟踪，dbms_monitor包分别提供了session_trace_enable和session_trace_disable过程。

以下PL/SQL调用针对ID为127、序列号为29的会话启用级别8的SQL跟踪：

```
dbms_monitor.session_trace_enable(session_id => 127,
                                  serial_num => 29,
                                  waits      => TRUE,
                                  binds      => FALSE,
                                  plan_stat  => 'first_execution')
```

所有参数都有默认值。如果有两个关于会话的参数没有指定，就针对执行此PL/SQL调用的会话启用SQL跟踪。

当通过session_trace_enable启用SQL跟踪时，也会相应地设置视图v$session中的sql_trace、sql_trace_waits和sql_trace_binds列。此外，自版本11.1开始，sql_trace_plan_stats列也会生效。注意，直至（并包括）10.2.0.5，这只会在以下情况下发生：在执行session_trace_enable后至少有一条SQL语句在被跟踪的会话中执行。例如，以下信息在执行了之前的PL/SQL调用后才能查询到：

3

```
SQL> SELECT sql_trace, sql_trace_waits, sql_trace_binds, sql_trace_plan_stats
  2  FROM v$session
  3  WHERE sid = 127;
SQL_TRACE SQL_TRACE_WAITS SQL_TRACE_BINDS SQL_TRACE_PLAN_STATS
--------- --------------- --------------- --------------------
ENABLED   TRUE            FALSE           FIRST EXEC
```

下面的PL/SQL调用禁用了ID为127、序列号为29的SQL跟踪：

```
dbms_monitor.session_trace_disable(session_id => 127,
                                   serial_num => 29)
```

请注意，这两个参数都有默认值。如果不指定，会禁用与执行这个PL/SQL调用对应的会话的SQL跟踪。

在RAC（Real Application Cluster，真实应用程序集）环境中，session_trace_enable和session_trace_diable需要在存在会话的对应数据库实例上执行。

● ***客户端级别***

为客户端启用和禁用SQL跟踪，dbms_monitor包分别提供了client_id_trace_enable和client_id_trace_disable过程。这些过程仅会在已设置会话属性的客户端标识符的情况下使用。

以下PL/SQL调用为所有具有指定客户端标识符的会话启用了级别12的SQL跟踪：

```
dbms_monitor.client_id_trace_enable(client_id => 'helicon.antognini.ch',
                                    waits     => TRUE,
                                    binds     => TRUE,
                                    plan_stat => 'first_execution')
```

参数client_id没有默认值，并且区分大小写。

由于这个设置会保存在数据字典里，所以实例重启后也会存在，同时，在一个RAC的环境下，它对所有数据库实例生效。

在dba_enabled_traces和12.1多租户环境下的cdb_enabled_traces视图里，会通过client_id_trace_enable过程，显示启用SQL跟踪的用户标识符以及使用的参数。例如，使用以上的PL/SQL调用启用SQL跟踪后，可以查到如下信息：

```
SQL> SELECT primary_id AS client_id, waits, binds, plan_stats
  2  FROM dba_enabled_traces
  3  WHERE trace_type = 'CLIENT_ID';

CLIENT_ID            WAITS BINDS PLAN_STATS
-------------------- ----- ----- ----------
helicon.antognini.ch TRUE  TRUE  FIRST_EXEC
```

以下PL/SQL调用针对所有指定客户端标识符的会话禁用SQL跟踪：

```
dbms_monitor.client_id_trace_disable(client_id => 'helicon.antognini.ch')
```

过程client_id_trace_disable移除过程client_id_trace_enable相应在数据字典里增加的信息。参数client_id没有默认值。

● **组件级别**

包dbms_monitor分别提供了过程serv_mod_act_trace_enable和serv_mod_act_trace_disable来利用服务名、模块名和动作名为组件启用和禁用SQL跟踪。要充分使用这些过程，你需要设置会话属性、模块名和动作名。

以下PL/SQL调用为所有使用指定参数的会话启用SQL跟踪：

```
dbms_monitor.serv_mod_act_trace_enable(service_name  => 'DBM11203.antognini.ch',
                                       module_name   => 'mymodule',
                                       action_name   => 'myaction',
                                       waits         => TRUE,
                                       binds         => TRUE,
                                       instance_name => NULL,
                                       plan_stat     => 'all_executions')
```

这里唯一一个没有默认值的参数是第一个service_name[1]。参数module_name和action_name默认值分别为any_module和any_action。同时，NULL也是一个有效的值。如果指定了参数action_name的值，那么也必须指定参数module_name的值。如果不设置，则会引发ORA-13859错误。在RAC环境下，参数instance_name用来限定具体跟踪哪个数据库实例。默认情况下，SQL跟踪对所有数据库实例生效。请注意，参数service_name、module_name、action_name和instance_name区分大小写。

由于设置保存在数据字典中，数据库实例重启不会影响使用。

与客户端级别的SQL跟踪相同，在dba_enabled_traces和12.1多租户环境下的cdb_enabled_traces视图里，通过过程serv_mod_act_trace_enable，会显示启用了SQL跟踪的用户标识符组件以及使用的参数。使用以上PL/SQL调用启用SQL跟踪后，你可以查询到以下信息：

```
SQL> SELECT primary_id AS service_name, qualifier_id1 AS module_name,
  2         qualifier_id2 AS action_name, waits, binds, plan_stats
  3  FROM dba_enabled_traces
  4  WHERE trace_type IN ('SERVICE', 'SERVICE_MODULE', 'SERVICE_MODULE_ACTION');

SERVICE_NAME          MODULE_NAME ACTION_NAME WAITS BINDS PLAN_STATS
--------------------- ----------- ----------- ----- ----- ----------
DBM10203.antognini.ch mymodule    myaction    TRUE  TRUE  ALL_EXEC
```

注意，根据启用SQL跟踪指定的参数定义（即服务名、模块名和动作名），会将列trace_type设置成SERVICE、SERVICE_MODULE或者SERVICE_MODLE_ACTION。

以下PL/SQL调用为所有使用指定参数的会话禁用SQL跟踪：

```
dbms_monitor.serv_mod_act_trace_disable(service_name  => 'DBM11203.antognini.ch',
                                        module_name   => 'mymodule',
                                        action_name   => 'myaction',
                                        instance_name => NULL)
```

---

① 服务名对于数据库来说是一个逻辑名。它是通过初始化参数service_names或包dbms_service进行配置的。一个数据库可以有多个服务名。

过程serv_mod_act_trace_disable会移除过程serv_mod_act_trace_enable添加到数据字典里的信息。所有参数都与过程serv_mod_act_trace_enable具有一样的默认值并且作用一致。

- **数据库级别**

包dbms_monitor为所有连接到数据库的会话启用和禁用SQL跟踪（那些后台进程创建的除外），分别提供过程database_trace_enable和database_trace_disable。

以下PL/SQL调用为单个数据库实例启用级别12的SQL跟踪：

```
dbms_monitor.database_trace_enable(waits         => TRUE,
                                   binds         => TRUE,
                                   instance_name => 'DBM11203',
                                   plan_stat     => 'first_execution')
```

所有参数都有默认值。在RAC环境下，使用参数instance_name来限制要跟踪的数据库实例。指定的值可以从视图gv$instance的列instance_name获得。如果将参数instance_name设置为NULL（同时也是默认值），SQL跟踪会在所有数据库实例启用。请注意，参数instance_name区分大小写。

由于设置保存在数据字典中，数据库实例重启不会影响使用。

与客户端级别和组件级别的SQL跟踪相同，在dba_enabled_traces和12.1多租户环境下的cdb_enabled_traces视图里，会通过过程database_trace_enable显示启用SQL跟踪的用户标识符以及使用的参数。例如，使用以上PL/SQL调用启用SQL跟踪后，可以查询到以下信息：

```
SQL> SELECT instance_name, waits, binds, plan_stats
  2  FROM dba_enabled_traces
  3  WHERE trace_type = 'DATABASE';

INSTANCE_NAME WAITS BINDS PLAN_STATS
------------- ----- ----- ----------
DBM11203      TRUE  TRUE  FIRST_EXEC
```

以下PL/SQL调用会禁用数据库级别的SQL跟踪，同时会移除过程database_trace_enable相应加入到数据字典里的信息：

```
dbms_monitor.database_trace_disable(instance_name => 'DBM11203')
```

请注意，这不会禁用其他级别（包括会话级别、客户端级别或组件级别）的SQL跟踪。如果将参数instance_name设置为NULL（同时也是默认值），则会禁用所有数据库实例的SQL跟踪功能。

3. 使用DBMS_SESSION启用SQL跟踪

之前曾指出，默认情况下访问包dbms_monitor是有限制的。如果想为当前连接的会话启用或禁用SQL跟踪，但你既没有包dbms_monitor的执行权限，又不想执行ALTER SESSION语句（比如，因为语法很难记），则可以使用包dbms_session。

包dbms_session包含两个过程：session_trace_enable和session_trace_disable，它们的功能与包dbms_monitor下的同名过程一致。唯一的区别就是，dbms_session下的过程只能为当前连接的会话启用或禁用SQL跟踪。因此，拥有执行ALTER SESSION语句权限的任何用户都可以使用这两个过程。

下面举例说明如何使用dbms_session启用和禁用SQL跟踪。注意视图v$session提供的输出表明SQL跟踪已经启用：

```
SQL> BEGIN
  2    dbms_session.session_trace_enable(waits     => TRUE,
  3                                      binds     => TRUE,
  4                                      plan_stat => 'all_executions');
  5  END;
  6  /

SQL> SELECT sql_trace, sql_trace_waits, sql_trace_binds, sql_trace_plan_stats
  2  FROM v$session
  3  WHERE sid = sys_context('userenv','sid');

SQL_TRACE SQL_TRACE_WAITS SQL_TRACE_BINDS SQL_TRACE_PLAN_STATS
--------- --------------- --------------- --------------------
ENABLED   TRUE            TRUE            ALL EXEC

SQL> BEGIN
  2    dbms_session.session_trace_disable;
  3  END;
  4  /
```

4. 触发SQL跟踪

在上面的内容中，你看到了启用和禁用SQL跟踪的不同方法。最简单的情况是手工执行SQL语句或PL/SQL调用。不过，有时自动触发SQL跟踪很必要。“自动”在这里表示代码必须加在某处。

最简单的方法是在数据库级别创建一个登录触发器。为了避免启用所有用户的SQL跟踪，我通常建议创建一个角色（在接下来的例子中命名为sql_trace），并暂时只把它赋予给需要启用SQL跟踪的用户。以下示例是脚本sql_trace_trigger.sql的节选：

```
CREATE ROLE sql_trace;

CREATE OR REPLACE TRIGGER enable_sql_trace AFTER LOGON ON DATABASE
BEGIN
  IF (dbms_session.is_role_enabled('SQL_TRACE'))
  THEN
    EXECUTE IMMEDIATE 'ALTER SESSION SET timed_statistics = TRUE';
    EXECUTE IMMEDIATE 'ALTER SESSION SET max_dump_file_size = unlimited';
    dbms_session.session_trace_enable;
  END IF;
END;
```

自然地，这也可以定义成针对单个架构的触发器或者基于其他执行的检查，例如，基于userenv命令。请注意，除了启用SQL跟踪之外，还可以做一些与SQL跟踪相关的初始参数设置的练习（更多内容会在本章后续部分介绍）。

注意　执行上面的触发器所需要的ALTER SESSION执行权限不能通过角色赋予，而是需要直接赋予给用户来创建此触发器。

另一个方法是在应用里直接添加代码来启用SQL跟踪。某些用来触发代码的参数也需要添加进去。胖客户端应用的命令行参数或者网页应用的附加HTTP参数就是一个很好的例子。

5. 跟踪文件里的定时信息

初始化参数timed_statistics控制着跟踪文件里的定时信息，比如运行时间和CPU时间，此参数可以设置成TRUE或FALSE。如果设置成TRUE，定时信息会保存到跟踪文件里。如果设置成FALSE，则相反。然而，根据工作平台的不同，部分平台下也会记录定时信息。timed_statistics的默认值取决于另外一个初始化参数：statistics_level。如果将statistics_level设置成basic，那么timed_statistics默认为FALSE。否则，timed_statistics默认为TRUE。

通常来说，如果定时信息不可用，那么跟踪文件是没用的。因此，在启用SQL跟踪前，请确保参数timed_statistics已设置为TRUE。比如，可以执行以下SQL语句：

```
ALTER SESSION SET timed_statistics = TRUE
```

动态初始化参数

初始化参数有静态的，也有动态的。如果是动态参数，代表可以改变它们而不用重启实例。在动态初始化参数中，有些参数仅可以在会话级别做更改，有些只可以在系统级别修改，其他参数两者均可。在会话和系统级别修改初始化参数，可以分别使用ALTER SESSION和ALTER SYSTEM语句。系统级别的初始化参数修改完会立刻生效或者仅对修改后创建的会话生效。v$parameter视图中，或者更准确地说是列isses_modifiable和issys_modifiable中，记录着初始化参数在哪些情况下可以进行修改。

6. 限制跟踪文件大小

通常，没有人会在意跟踪文件的大小限制。如果必须要限制其大小，可以在会话或者系统级别设置初始化参数max_dump_file_size。指定具体数值后跟K或者M，代表KB或者MB，以此表示跟踪文件的最大文件大小。如果希望没有限制，可以像以下SQL这样，设置初始化参数的值为unlimited：

```
ALTER SESSION SET max_dump_file_size = 'unlimited'
```

从版本11.1开始，当达到限制时，会在告警日志里记录如下信息：

```
Non critical error ORA-48913 caught while writing to trace file "/u00/app/oracle/diag/rdbms/
dbm11203/DBM11203/trace/DBM11203_ora_6777.trc"
Error message: ORA-48913: Writing into trace file failed, file size limit [512000] reached
Writing to the above trace file is disabled for now on...
```

7. 找到跟踪文件

跟踪文件是由在数据库服务器上运行的数据库引擎服务器进程创建的。这表明跟踪文件是由数据库服务器直接写入到硬盘。在版本10.2中，不同进程产生的跟踪文件会写入不同的目录。

- 专用服务器进程创建的跟踪文件会写入初始化参数user_dump_dest指定的位置。
- 后台进程创建的跟踪文件会写入初始化参数background_dump_dest指定的位置。

进程类型可以通过视图v$session下的type字段来辨别。但奇怪的是，并不是所有后台进程都可以在视图v$bgprocess中查到。

从版本11.1开始，随着自动诊断信息库（ADR）的引入，初始化参数user_dump_dest和background_dump_dest已被初始化参数diagnostic_dest所取代。由于新的初始化参数只设置了基础目录，因此你可以查询视图v$diag_info来获取跟踪文件的准确位置。以下查询列举出了初始化参数与跟踪位置的

区别：

```
SQL> SELECT value FROM v$parameter WHERE name = 'diagnostic_dest';

VALUE
---------------
/u00/app/oracle

SQL> SELECT value FROM v$diag_info WHERE name = 'Diag Trace';

VALUE
-------------------------------------------------
/u00/app/oracle/diag/rdbms/dbm11203/DBM11203/trace
```

请注意，在12.1多租户环境中，不能在PDB级别设置初始化参数user_dump_dest、background_dump_dest和diagnostic_dest。

跟踪文件以前曾根据版本和平台来命名。在最近的版本中，命名是根据以下结构：

```
{instance_name}_{process_name}_{process_id}.trc
```

下面是该结构的分解说明。

- instance_name：这是初始化参数instance_name的值。请注意，尤其在RAC环境中，初始化参数instance_name与初始化参数db_name是不同的。初始化参数instance_name可以在视图gv$instance的列instance_name中查到。
- process_name：这是产生跟踪文件的进程的小写名称。对于专用服务器进程，会用ora作为进程名。对于共享服务器进程，进程名来自视图v$dispatcher或v$shared_server的name列。对于并行从属进程，进程名根据视图v$px_process的server_name列来命名。对于其他大部分后台进程，进程名根据视图v$bgprocess的name列来命名。
- process_id：系统级别的进程标识符（Windows下是线程标识符）。这个值可以在视图v$process的spid列中找到。

根据这里提供的信息，我们可以写出一个类似脚本map_session_to_tracefile.sql中的语句。但这样的语句只能在10.2版本中使用。从11.1版本开始，就像下面给出的例子，只需要查询视图v$diag_info或者v$process就可以：

```
SQL> SELECT value
  2  FROM v$diag_info
  3  WHERE name = 'Default Trace File';

VALUE
--------------------------------------------------------------------
/u00/app/oracle/diag/rdbms/dba111/DBA111/trace/DBA111_ora_23731.trc

SQL> SELECT p.tracefile
  2  FROM v$process p, v$session s
  3  WHERE p.addr = s.paddr
  4  AND s.sid = sys_context('userenv','sid');

TRACEFILE
--------------------------------------------------------------------
/u00/app/oracle/diag/rdbms/dba111/DBA111/trace/DBA111_ora_23731.trc
```

注意，视图V$diag_info只为当前对话提供信息。

跟踪文件会包含机密信息吗

默认情况下，不是任何用户都可以访问跟踪文件，这样做的好处是因为跟踪文件里可能包含机密信息。实际上，包含文字的SQL语句和绑定变量的值都会被记录在跟踪文件里。这表明任何存储在数据库里的数据也都可以被写入跟踪文件。

例如，在Unix/Linux数据库服务器上，跟踪文件属于运行数据库引擎二进制文件的用户和组，并且默认情况下它拥有0640权限。换句话说，只有与运行数据库引擎的用户处于同一组中的用户才可以读取跟踪文件。

因此，如果那些能够访问数据库的用户需要执行任务，就没有什么理由阻止他们去访问跟踪文件。实际上，从安全的角度看，跟踪文件仅对于那些无权访问数据库的用户来说才是有用的信息源。为此，数据库引擎提供了一个未公开的初始化参数_trace_files_public。默认情况下，它被设置为FALSE。如果设置成TRUE，那么跟踪文件会对所有具有访问数据库权限的用户公开。此初始化参数不是动态的，因此，修改需要重启实例。请注意，在12.1多租户环境下，不可以在PDB级别设置此参数。

比如，在Unix/Linux下，把_trace_files_public设置为TRUE，那么默认的权限会变为0644。这样，所有能访问数据库的用户都可以访问跟踪文件。

从安全的角度看，只有当访问数据库不受限时，把初始化参数_trace_files_public设置成TRUE才会成为问题。为了方便访问跟踪文件，常见的做法是通过SMB或NFS共享目录，或者通过HTTP接口来实现。无论如何，每次需要跟踪文件时都让DBA手工传一份，这样可以最大程度地避免很多问题。

使用初始化参数tracefile_identifier也可能轻松找到想要的跟踪文件。实际上，使用这个初始化参数可以为跟踪文件的命名增加最多255个字符的自定义标识。添加标识的跟踪文件名结构会变成以下这样：

```
{instance_name}_{process name}_{process id}_{tracefile _identifier}.trc
```

初始化参数tracefile_identifier只可以在会话级别设置，并且只能是专用服务器进程。需要注意的是，每当一个会话动态修改了该参数，都会自动创建一个新的跟踪文件。可以在视图v$process的列traceid中找到初始化参数tracefile_identifier的值。请注意，在10.2版本中，该参数只对同样的会话生效，其他会话看到的参数值都是NULL。

现在我们知道了什么是SQL跟踪，如何配置、启用和禁用它，以及在哪里能找到生成的跟踪文件。下面来讨论一下它的结构和用来分析的工具，以及因此能看到的跟踪文件的内容。

3.1.2 跟踪文件的结构

跟踪文件包含特定进程执行数据库调用的信息。实际上，当一个进程ID在操作系统级别被重用时，会导致一个跟踪文件里包含多个进程信息。因为不同的会话可能会使用同一个进程（比如共享服务器

或者并行从属进程）并且每个会话都有不同的会话属性（比如模块名和动作名），所以跟踪文件会被分成多个逻辑部分。请看图3-3的例子（这两个跟踪文件和本章的其他文件都可供下载）。

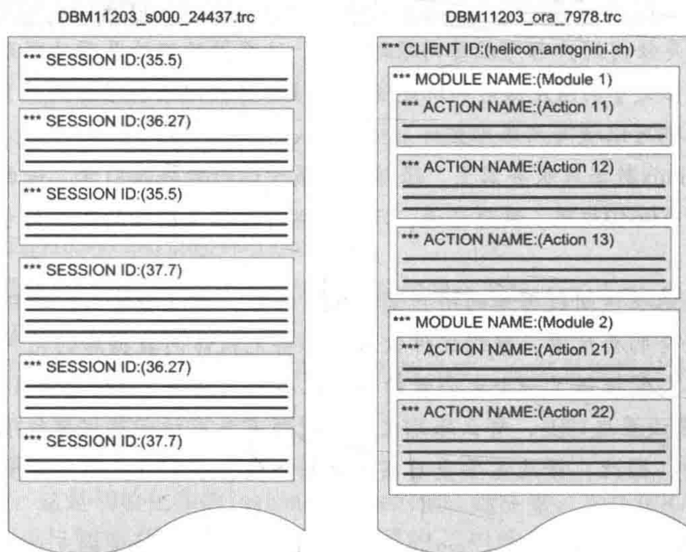


图3-3　跟踪文件可以由多个逻辑段落组成。左边是一个共享服务器的跟踪文件，包含三个会话信息。右边是专用服务器的跟踪文件，包含一个客户端的两个模块和五个动作信息

在图3-3右边显示的跟踪文件结构可以使用之前提供的PL/SQL块来开启SQL跟踪：

```
DECLARE
  l_dummy VARCHAR2(10);
BEGIN
  dbms_session.set_identifier(client_id => 'helicon.antognini.ch');
  dbms_application_info.set_module(module_name => 'Module 1',
                                   action_name => 'Action 11');
  -- code module 1, action 11
  SELECT 'Action 11' INTO l_dummy FROM dual;
  dbms_application_info.set_module(module_name => 'Module 1',
                                   action_name => 'Action 12');
  -- code module 1, action 12
  SELECT 'Action 12' INTO l_dummy FROM dual;
  dbms_application_info.set_module(module_name => 'Module 1',
                                   action_name => 'Action 13');
  -- code module 1, action 13
  SELECT 'Action 13' INTO l_dummy FROM dual;
  dbms_application_info.set_module(module_name => 'Module 2',
                                   action_name => 'Action 21');
  -- code module 2, action 21
  SELECT 'Action 21' INTO l_dummy FROM dual;
  dbms_application_info.set_module(module_name => 'Module 2',
                                   action_name => 'Action 22');
  -- code module 2, action 22
  SELECT 'Action 22' INTO l_dummy FROM dual;
END;
```

在图3-3中，在跟踪文件里由三个星号（***）开头的标签用来标识段落。跟踪文件之间的区别不光是数据库引擎会在每个部分重复记录一些信息，另外还会加上时间戳。下面是由PL/SQL块生成的跟踪文件的片段内容：

```
*** CLIENT ID:(helicon.antognini.ch) 2012-11-30 10:05:05.531
*** MODULE NAME:(Module 1) 2012-11-30 10:05:05.531
*** ACTION NAME:(Action 11) 2012-11-30 10:05:05.531
...
...
*** MODULE NAME:(Module 1) 2012-11-30 10:05:05.532
*** ACTION NAME:(Action 12) 2012-11-30 10:05:05.532
...
...
*** MODULE NAME:(Module 1) 2012-11-30 10:05:05.533
*** ACTION NAME:(Action 13) 2012-11-30 10:05:05.533
...
...
*** MODULE NAME:(Module 2) 2012-11-30 10:05:05.533
*** ACTION NAME:(Action 21) 2012-11-30 10:05:05.533
...
...
*** MODULE NAME:(Module 2) 2012-11-30 10:05:05.534
*** ACTION NAME:(Action 22) 2012-11-30 10:05:05.534
```

这些逻辑会话标签很有用。有了它们，才可以根据你的需要来提取相关信息。例如，如果你关注的性能问题与一个特殊动作有关，就可以把跟踪文件里相关的部分独立出来。可以使用工具TRCSESS来实现，下面我们来介绍一下。

3.1.3 使用 TRCSESS

可以根据之前介绍的逻辑部分来使用命令行工具TRCSESS，从一个或者多个跟踪文件中提取需要的信息。如果运行TRCSESS时未指定任何参数作为输入，那么返回的就是完整的TRCSESS参数列表，其中包括简短的描述。

```
trcsess [output=<output file name >] [session=<session ID>] [clientid=<clientid>]
        [service=<service name>] [action=<action name>] [module=<module name>]
        <trace file names>

output=<output file name> output destination default being standard output.
session=<session Id> session to be traced.
Session id is a combination of session Index & session serial number e.g. 8.13.
clientid=<clientid> clientid to be traced.
service=<service name> service to be traced.
action=<action name> action to be traced.
module=<module name> module to be traced.
<trace_file_names> Space separated list of trace files with wild card '*' supported.
```

正如你所见，可以将会话、客户端标识符、服务名称、模块名称和动作名称指定为参数。例如，要从跟踪文件DBM11203_ora_7978.trc中提取关于Action 12的信息，并写入到一个名为action12.trc的新文件中，可以使用下面的命令：

```
trcsess output=action12.trc action="Action 12" DBM11203_ora_7978.trc
```

请注意，参数clientid、service、action和module都是区分大小写的。

3.1.4 探查器

一旦你定位到正确的跟踪文件，或者使用TRCSESS提取出了需要的部分，就开始着手分析内容吧。你需要使用探查器（profiler），目的是基于原始跟踪文件的内容生成一份格式化输出。Oracle的服务端和客户端都包含此工具，叫作TKPROF（Trace Kernel PROFiler）。即使工具输出的结果对有些状况有帮助，有时也并不足以胜任快速定位性能问题的工作。奇怪的是，Oracle低估了此工具的重要性，从Oracle7开始引入此工具后就很少改进它。现在市面上有一些商业的和免费的探查器。我自己也开发了一个免费的探查器TVD$XTAT。你也可以考虑使用其他的探查器：OraSRP[1]、Method R Profiler和Method R Tools suite[2]。甚至Oracle都建议（通过Oracle支持）使用另一个称为Trace Analyzer[3]的探查器。

接下来的两节会介绍其中的两个探查器。首先是TKPROF，尽管它有很多不足，却是唯一一个在任何情况下都能使用的探查器。实际上，很多时候你不能在数据库服务器上安装其他探查器或者把跟踪文件下载到其他机器上。在这样的情况下，TKPROF是很有用的。然后我会介绍自己写的探查器。这里会用到使用以下PL/SQL块生成的跟踪文件：

```
DECLARE
  l_count INTEGER;
BEGIN
  FOR c IN (SELECT extract(YEAR FROM d), id, pad
            FROM t
            ORDER BY extract(YEAR FROM d), id)
  LOOP
    NULL;
  END LOOP;
  FOR i IN 1..10
  LOOP
    SELECT count(n) INTO l_count
    FROM t
    WHERE id < i*123;
  END LOOP;
END;
```

3.1.5 使用 TKPROF

TKPROF是命令行工具，它的主要作用是输入一个原始的跟踪文件并输出一个格式化后的文本文件。此工具还可以生成SQL脚本以在数据库中加载数据，尽管这个特性很少有人使用。

仅通过指定一个输入文件和输出文件可以执行最简单的分析。在下面的例子中，输入文件是DBM11106_ora_6334.trc，输出文件是DBM11106_ora_6334.trc。

```
tkprof DBM11106_ora_6334.trc DBM11106_ora_6334.txt
```

---

① 具体信息请访问http://www.oracledba.ru/orasrp。

② 具体信息请访问http://method-r.com。

③ 更多信息请查看Oracle Support文档*TRCANLZR (TRCA): SQL_TRACE/Event 10046 Trace File Analyzer-Tool for Interpreting Raw SQLTraces*（224270.1）。

尽管默认的输出文件扩展名是prf，我个人还是喜欢使用txt。在我看来，使用扩展名可以让所有人明白文件类型，同时可以在任何操作系统中正确识别出来。

未附加其他参数的分析仅对非常小的跟踪文件有用。在大多数情况下，你需要指定多个参数来获得一个更好的输出文件。

1. TKPROF参数

如果运行TKPROF时未附加参数，返回的就是完整的TRCSESS参数列表，其中包含简短的描述。

```
Usage: tkprof tracefile outputfile [explain= ] [table= ]
              [print= ] [insert= ] [sys= ] [sort= ]
  table=schema.tablename   Use 'schema.tablename' with 'explain=' option.
  explain=user/password    Connect to ORACLE and issue EXPLAIN PLAN.
  print=integer    List only the first 'integer' SQL statements.
  aggregate=yes|no
  insert=filename  List SQL statements and data inside INSERT statements.
  sys=no           TKPROF does not list SQL statements run as user SYS.
  record=filename  Record non-recursive statements found in the trace file.
  waits=yes|no     Record summary for any wait events found in the trace file.
  sort=option      Set of zero or more of the following sort options:
    prscnt  number of times parse was called
    prscpu  cpu time parsing
    prsela  elapsed time parsing
    prsdsk  number of disk reads during parse
    prsqry  number of buffers for consistent read during parse
    prscu   number of buffers for current read during parse
    prsmis  number of misses in library cache during parse
    execnt  number of execute was called
    execpu  cpu time spent executing
    exeela  elapsed time executing
    exedsk  number of disk reads during execute
    exeqry  number of buffers for consistent read during execute
    execu   number of buffers for current read during execute
    exerow  number of rows processed during execute
    exemis  number of library cache misses during execute
    fchcnt  number of times fetch was called
    fchcpu  cpu time spent fetching
    fchela  elapsed time fetching
    fchdsk  number of disk reads during fetch
    fchqry  number of buffers for consistent read during fetch
    fchcu   number of buffers for current read during fetch
    fchrow  number of rows fetched
    userid  userid of user that parsed the cursor
```

每个参数的功能如下。

- explain会使TKPROF为跟踪文件中的每个SQL语句生成执行计划。实现方式是通过执行EXPLAIN PLAN语句产生的结果（关于此SQL语句的详细信息请看第10章）。很显然，为了执行SQL语句，就必须连接数据库。因此，此参数用来指定用户名、密码和连接字符串。可供使用的格式是explain=user/password@connect_string和explain=user/password。请注意，为了能够尽最大可能得到正确的执行计划，你应该使用与生成跟踪文件时相同的用户，并且确保所有查询优化器的初始化参数也与生成跟踪文件时相同。同时也要注意初始化参数会随着程序运行时或登录触

发器而更改。最好的情况就是你能使用相同的用户，但无论如何，即使所有条件都满足，因为使用EXPLAIN PLAN语句生成的执行计划不一定与真正的执行计划一致（原因会在第10章解释），所以不建议指定explain参数。如果指定了错误的用户名、密码或连接字符串，则会分析跟踪文件而不返回任何错误信息。反之，则会在输出文件的头部找到类似以下的错误信息：

```
error connecting to database using: scott/lion
ORA-01017: invalid username/password; logon denied
EXPLAIN PLAN option disabled.
```

- table只可以和explain参数一起使用。它的作用实际上是指定哪张表使用EXPLAIN PLAN语句生成执行计划。通常可以不指定此参数，因为TKPROF会自动创建并删除一个名为prof$plan_table的计划表用做分析。总之，如果用户无法创建表（比如没有CREATE TABLE权限），就必须指定table参数。例如，要指定system用户下的表plan_table，那么参数必须设置成table=system.plan_table。执行分析的用户必须对特定的表具有SELECT、INSERT和DELETE权限。同样，错误也只会记录在输出文件里。
- print用来限制输出文件里SQL语句的行数。默认情况下没有限制。这个参数只有与参数sort一起使用才有意义，用来输出top SQL语句。例如，为了只获得10条SQL语句，参数必须设置成print = 10。
- aggregate指定TKPROF该如何处理相同的SQL语句。默认情况下（aggregate=yes），所有指定SQL语句的信息都会进行汇总。11.2版本中又进一步要求执行计划也要相同才可以。因此，在11.2版本中，默认会汇总对应SQL语句的每条执行计划信息。汇总信息会独立于跟踪文件里的SQL语句，因此就汇总来说，信息会缺少一部分。即使在许多情况下默认设置可以满足分析，但有时最好设置aggregate=no来检查单独的SQL语句。
- TKPROF使用参数insert生成可以存储进数据库的SQL脚本。脚本名直接用参数自身指定，比如insert = load.sql。
- sys参数指定由sys用户执行的SQL语句（典型情况下，解析操作需要递归查询数据字典）是否要写入输出文件。默认值是yes，但大多数时候我更愿意设置成no来避免无用的信息写入输出文件。你通常无法控制sys用户递归执行SQL语句，因此这是多余的。
- TKPROF使用参数record生成跟踪文件里所有非递归语句的SQL脚本。脚本名称直接由参数指定（例如，record = replay.sql）。根据文档，这个特性可以用来手工重播SQL语句，但由于不会处理绑定变量，这通常无法实现。
- waits确定是否将等待事件信息加入输出文件。默认情况下是加入的。就个人而言，输出文件里的等待事件非常重要，我认为不应该指定waits = no。
- sort指定写入输出文件的SQL语句顺序。默认情况下，是根据在跟踪文件里的读取顺序排序的。基本上，你可以根据资源利用（比如调用数、CPU时间和物理读）或响应时间（即运行时间）来对输出进行排序。如你所见，对于大多数选项（比如运行时间），每种类型的数据库调用的值都可用来排序：比如，解析游标所花费的时间prsela，执行游标所花费的时间exeela，以及从一个游标里获取数据所花费的时间fchela。尽管你可以根据多种选择和组合来进行排序，但对研究性能问题来说只有一种排序是真正有用的：response time。因此，你应该指定sort = prsela,exeela,fchela。TKPROF支持在参数中传入多个值，只需要用逗号分隔即可，甚至可

以传入不同单位的值。请注意，当跟踪文件里包含多个会话并且指定了参数aggregate=no，就会分别对每个会话的SQL语句进行排序。

基于以上信息，我个人常用的TKPROF参数如下：

```
tkprof {input trace file} {output file} sys=no sort=prsela,exeela,fchela
```

现在你知道如何使用TKPROF了，让我们来看看它生成的输出文件。

2. 解释TKPROF输出

分析的输出文件根据以下参数生成：

```
tkprof DBM11203_ora_28030.trc DBM11203_ora_28030.txt
      sort=prsela,exeela,fchela print=4 explain=chris/ian aggregate=no
```

请注意，这里不是建议你根据以上参数这么做，只是为了向你展示一个详细的输出。跟踪文件和输出文件同本章其他文件一起可供下载。

输出文件的头部信息大部分是静态的。然而，这里面也有有用的信息：跟踪文件名、生成输出文件时参数sort的值以及定位跟踪会话的一行信息。最后这条信息只有在指定了参数aggregate=no时才会显示。请注意，当跟踪文件包含多个会话并且指定参数aggregate=no时，头部信息里会反复使用分隔符来区别SQL语句和其他会话。

```
TKPROF: Release 11.2.0.3.0 - Development on Fri Nov 30 23:45:57 2012

Copyright (c) 1982, 2011, Oracle and/or its affiliates.  All rights reserved.
Trace file: DBM11203_ora_28030.trc
Sort options: prsela exeela fchela
********************************************************************************
count    = number of times OCI procedure was executed
cpu      = cpu time in seconds executing
elapsed = elapsed time in seconds executing
disk     = number of physical reads of buffers from disk
query    = number of buffers gotten for consistent read
current = number of buffers gotten in current mode (usually for update)
rows     = number of rows processed by the fetch or execute call
--------------------------------------------------------------------------------

*** SESSION ID: (156.29) 2012-11-30 23:21:45.691
```

任何连接数据库或者生成执行计划时的出错信息都会写在头部信息之后。

头部信息之后就是每条SQL语句的信息：SQL语句的文本、执行统计、解析信息、执行计划以及等待事件。仅当执行计划和等待事件存储在跟踪文件中时，才会报告执行计划和等待事件。请记住，在10.2版本中，只有当相关游标关闭后，才会将执行计划写入跟踪文件。这表明如果应用重用游标而不关闭它们，那么就不会将这些重用游标的执行计划写入跟踪文件。

SQL语句的文本在有些情况下是格式化后的。然而并不是所有情况下都能得到正确的显示格式。比如，在下面的例子中extract函数的关键字FROM就与SELECT语句的FROM子句混淆。请注意，SQL ID只对11.1.0.6及以后的版本有效，执行计划的值仅对11.1.0.7及以后的版本有效。

```
SQL ID: 7wd0gdwwgph1r Plan Hash: 961378228

SELECT EXTRACT(YEAR
```

```
FROM
 D), ID, PAD FROM T ORDER BY EXTRACT(YEAR FROM D), ID
```

执行统计会根据数据库调用的类型进行汇总并以表格形式显示。每一项性能指标如下所示。

- count：数据库调用执行的次数。
- cpu：花费在数据库调用上的CPU时间总和，单位秒。
- elapsed：花费在数据库调用上的运行时间总和，单位秒。如果此值高于CPU时间，那么在执行统计下面的部分你会找到等待事件，那里有资源或同步点等待的信息。
- disk：代表物理读的块数。注意，这不是物理I/O数。如果这个值比逻辑读大（disk>query + current），就代表块涌进了临时表空间。这种情况下你可以看到至少读取了33 017（71 499–38 474–8）个块。这需要稍后通过Row Source Operation的统计和等待事件来确认。
- query：一致性逻辑读的块数。通常查询会用到这种逻辑读。
- current：在当前模式下使用逻辑读读取的块数。通常INSERT、DELETE、MERGE和UPDATE语句修改块会产生此类逻辑读。
- rows：处理的行数。对于查询来说，这是获取的行数。对于INSERT、DELETE、MERGE和UPDATE来说这是受影响的行数。在这里10 001次调用获取了1 000 000行是没有任何意义的。这表示平均来看每次调用获取了大约100行。注意这里的100是在PL/SQL里设置的预获取值（第15章会介绍关于预获取值的详细信息）。

```
call     count       cpu    elapsed       disk      query    current       rows
------- ------  -------- ---------- ---------- ---------- ---------- ----------
Parse        1      0.00       0.00          0          0          0          0
Execute      1      0.00       0.00          0          0          0          0
Fetch    10001      6.49      11.92      71499      38474          8    1000000
------- ------  -------- ---------- ---------- ---------- ---------- ----------
total    10003      6.49      11.92      71499      38474          8    1000000
```

接下来的几行是关于解析的基本概括信息。头两个值（Misses in library cache）代表在解析和执行调用期间的硬解析数。如果在执行调用期间没有硬解析，那么这行就不会显示。接下来是优化器模式和解析此SQL语句的用户ID。请注意用户名（本例里是chris）只有指定了参数explain时才会显示。否则只会显示用户ID（这里是34）。最后一条信息是递归深度。这条信息仅是为递归SQL语句提供的。直接由应用执行的SQL语句深度为0。深度*n*（本例是1）仅代表另一个深度为*n*–1的SQL语句执行了这个SQL。在这个例子中，深度为0的SQL语句是由SQL*Plus执行的PL/SQL块。

```
Misses in library cache during parse: 1
Misses in library cache during execute: 1
Optimizer mode: ALL_ROWS
Parsing user id: 34    (CHRIS)         (recursive depth: 1)
```

在解析的基本信息之后就是执行计划了。实际上，如果指定了参数explain，可能会看到两部分信息。第一部分叫作Row Source Operation，是由多个服务器进程写入跟踪文件的执行计划。第二部分叫作Execution Plan，当指定参数explain后由TKPROF生成。由于它是后生成的，即使与第一部分不同也没关系。总之，如果你发现两部分不同，就以第一部分为准。

第10章会介绍如何阅读执行计划，这里只介绍TKPROF的细节。对于跟踪文件里的第一个执行计划，两部分执行计划都会有执行计划里每步执行返回的行数（注意，不是处理的行数）。除此之外，

11.2.0.2及更高的版本也提供了所有执行返回的平均行数和最大行数。执行计划本身的数量是由Number of plan statistics captured值提供的。

对于每个row source operation，还可能会提供以下运行时统计信息。

- cr是一致性逻辑读的块数。
- pr是物理读的块数。
- pw是物理写的块数。
- time是执行操作运行时间总和，单位微秒。请注意这里显示的值并不总是那么精确。因为为了降低开销，服务进程会使用采样来衡量。
- cost是操作的估计成本。这个值自11.1版本起开始启用。
- size是操作返回的预计数据大小（bytes）。这个值自11.1版本起开始启用。
- card是操作返回的预估行数。这个值自11.1版本起开始启用。

```
Number of plan statistics captured: 1

Rows (1st)  Rows (avg)  Rows (max)  Row Source Operation
----------  ----------  ----------  ---------------------------------------------------
   1000000     1000000     1000000  SORT ORDER BY (cr=38474 pr=71499 pw=33035 time=11123996 us
                                               cost=216750 size=264000000 card=1000000)
   1000000     1000000     1000000   TABLE ACCESS FULL T (cr=38474 pr=38463 pw=0 time=5674541 us
                                                     cost=21 size=264000000 card=1000000)
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   MODE: ALL_ROWS
1000000   SORT (ORDER BY)
1000000    TABLE ACCESS   MODE: ANALYZED (FULL) OF 'T' (TABLE)
```

请注意，除了查询优化器的估值，其他的运行时统计信息都是累加出来的，它包含了child row source operation的值。例如，SORT ORDER BY命令从临时表空间读取了33 036（71 499–38 463）个块。根据之前的执行信息（参见关于disk列部分的介绍），你应该能够估算出至少读取了33 017个块。同时请注意，虽然在以前的版本中这些值只跟第一次查询有关，但从11.2.0.2版本开始它们变成了所有查询的平均值；在这种情况下就变得没有区别了，因为只有一次查询。

接下来的部分简要说明了SQL语句等待的等待事件。针对每种类型的等待事件，会提供以下值。

- Times Waited是等待事件发生的次数。
- Max.Wait是单个等待事件等待的最大时间，单位秒。
- Toal Waited是等待事件的总等待时间，单位秒。

```
Elapsed times include waiting on following events:

Event waited on                             Times Waited  Max. Wait  Total Waited
------------------------------------------  ------------  ---------  ------------
db file sequential read                                2       0.00          0.00
db file scattered read                               530       0.06          2.79
direct path write temp                             11002       0.00          0.51
direct path read temp                              24015       0.00          2.41
```

理想情况下，所有等待事件的等待时间总和应该与执行统计信息里的运行时间与CPU时间的差值相等。这个差值称为未被计算的时间。

未被计算的时间

SQL跟踪会提供每次操作执行时数据库所花费时间的信息。理想情况下，计算应该非常精确。但是，几乎不能在跟踪文件里找到精确到秒级以下的准确信息。当真实的运行时间与跟踪文件里记录的时间不同时，就存在**未被计算的时间**。

```
unaccounted-for time = real elapsed time - accounted for time
```

出现未被计算的时间最常见的原因如下。

- 最明显的是跟踪文件里缺少定时信息或等待事件。前者是因为初始化参数timed_statistics被设置成了FALSE。后者是因为启动SQL跟踪未包含级别8。在这两种情况下，未被计算的时间总是正值。正常情况下，恰当开启扩展的SQL跟踪能够避免这些问题。
- 通常来说，进程有三种状态：在CPU上运行、等待请求（例如，执行磁盘I/O操作）或在执行队列等待CPU资源。植入代码能够计算出前两种状态所花费的时间，但是对在执行队列等待多久却无能为力。因此，如果发生了CPU匮乏，那么未被计算的时间（始终为正值）可能会很长。基本上你可以通过两种方法避免这个问题：增加可用CPU的数量或者降低CPU使用率。
- 植入代码可做出精确的时间测量。然而，由于计算机系统中计时器的影响，每次测量会产生很小的量化误差。尤其是当有大量的测量时间时，这种量化误差造成的未被计算的时间会很明显。由于计时器自身的性质，量化误差可能会导致未被计算的时间为正值，也可能导致其为负值。遗憾的是，你对此无能为力。在实际应用中，这个问题与大量未被计算的时间无关，因为正误差往往会与负误差相抵消。
- 如果你排除了列出的这些方面，那么原因很可能是检测代码没有涵盖整个代码。例如，写入跟踪文件的时间就不会算在内。这通常不会是个问题。但如果跟踪文件写入一个性能差的设备或需要生成跟踪的信息量非常大，这时可能会产生大的开销。在这种情况下，未被计算的时间将始终为正值。为了避免这个问题，你应该把跟踪文件写入一个可以支撑正常吞吐量的设备上。在一些罕见的情况下，你或许会强制把跟踪文件放到RAM盘上。

由于等待事件的值已经高度聚合，这可以让你只需要知道你在等待哪类资源。比如，根据之前的信息，几乎所有的等待时间都花在了物理读上，但由于信息已经聚合，我们看不到具体信息，比如是从哪个数据文件读取的（原始跟踪文件里有这个信息）。实际上单块读的等待事件是db file sequential read，而多块读（参见第9章）的等待事件是db file scattered read。另外，涌入临时表空间的等待事件是direct path write temp和direct path read temp。

在分析等待事件时，关键是弄清楚与什么操作相关。幸运的是，即使等待事件种类有几百种，经常遇到的也就那么几种。在*Oracle Database Reference*手册的附录中有大部分等待事件的简介。

我们继续分析下一个SQL语句。由于这些信息结构与之前的一样，因此这里仅会注释输出文件中的新内容或本质的区别。

```
DECLARE
  l_count INTEGER;
BEGIN
  FOR c IN (SELECT extract(YEAR FROM d), id, pad
            FROM t
```

```
            ORDER BY extract(YEAR FROM d), id)
    LOOP
      NULL;
    END LOOP;
    FOR i IN 1..10
    LOOP
      SELECT count(n) INTO l_count
      FROM t
      WHERE id < i*123;
    END LOOP;
  END;
```

PL/SQL块的执行统计信息是有限的。它缺少物理读和逻辑读的信息，这是因为递归SQL语句（比如之前分析过的查询）使用的资源与父SQL语句无关。这表示你仅能看到SQL语句（或者PL/SQL块）本身使用的资源。

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ---------- ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.44       0.40          0          0          0           1
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ---------- ----------
total        2      0.45       0.41          0          0          0           1
```

由于PL/SQL块不是由数据库递归执行的，因此这里不会显示递归深度（递归深度为0）。同样，也不会显示执行计划。

```
Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 34     (CHRIS)
```

当网络层发送数据给客户端时（注意，通过网络发送数据的时间不会计算在内），数据库等待SQL*Net message to client，而数据库等待客户端返回消息时等待SQL*Net message from client。因此，对于每个由SQL*Net层完成的往返，你都能看到一对等待事件。请注意，在低级别层实现的往返次数会有些不同。比如，在网络层（比如IP）由于较小的数据包而完成大量往返的情况并不少见。

```
Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  SQL*Net message to client                       1        0.00          0.00
  SQL*Net message from client                     1        0.00          0.00
```

接下来是第二个由PL/SQL块执行的SQL语句。结构信息与之前的一致。需要指出的是，这个查询被执行了10次。对于每次执行，跟踪文件都包含执行统计信息（启用了级别16的）。因此，plan statistics captured的值是10，三行数据中每列都包含了不同的值（122 676和1229），并且行来源级别的运行时统计信息为平均值（比如，53的磁盘读除以10次执行，平均值为5）。

```
SQL ID: 7fjjjf0yvd05m Plan Hash: 4270555908

SELECT COUNT(N)
FROM
```

```
 T WHERE ID < :B1 *123

call     count       cpu    elapsed       disk      query    current        rows
------- ------ -------- ---------- ---------- ---------- ---------- ----------
Parse        1      0.00       0.00          0          0          0          0
Execute     10      0.00       0.00          0          0          0          0
Fetch       10      0.00       0.02         53        303          0         10
------- ------ -------- ---------- ---------- ---------- ---------- ----------
Total       21      0.01       0.02         53        303          0         10

Misses in library cache during parse: 1
Misses in library cache during execute: 1
Optimizer mode: ALL_ROWS
Parsing user id: 34  (CHRIS)   (recursive depth: 1)
Number of plan statistics captured: 10

Rows (1st)  Rows (avg)  Rows (max)  Row Source Operation
----------  ----------  ----------  ---------------------------------------------------
         1           1           1  SORT AGGREGATE (cr=30 pr=5 pw=0 time=2607 us)
       122         676        1229   TABLE ACCESS BY INDEX ROWID T (cr=30 pr=5 pw=0 time=2045 us
                                                               cost=8 size=1098 card=122)
       122         676        1229    INDEX RANGE SCAN T_PK (cr=4 pr=0 pw=0 time=872 us cost=3
                                                        size=0 card=122)(object id 20991)
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT MODE: ALL_ROWS
      1   SORT (AGGREGATE)
    122    TABLE ACCESS   MODE: ANALYZED (BY INDEX ROWID) OF 'T' (TABLE)
    122     INDEX   MODE: ANALYZED (RANGE SCAN) OF 'T_PK' (INDEX (UNIQUE)
                )

Elapsed times include waiting on following events:
  Event waited on                             Times Waited   Max. Wait  Total Waited
  ----------------------------------------   ------------  ----------  ------------
  db file sequential read                              53        0.00          0.02
```

为了获取被使用的对象信息（比如对象统计信息），数据库引擎递归执行最后的SQL语句。此外，查询优化器会使用此信息来计算出最有效率的执行计划。解析这条SQL的用户是SYS，因此你可以判断是由数据库引擎执行的这条SQL语句。根据递归深度为2，可以推断这条SQL语句需要解析深度为1的SQL，比如在输出文件中的第一个SQL语句。

```
SQL ID: 96g93hntrzjtr Plan Hash: 2239883476

select /*+ rule */ bucket_cnt, row_cnt, cache_cnt, null_cnt, timestamp#,
  sample_size, minimum, maximum, distcnt, lowval, hival, density, col#,
  spare1, spare2, avgcln
from
 hist_head$ where obj#=:1 and intcol#=:2


call     count       cpu    elapsed       disk      query    current        rows
------- ------ -------- ---------- ---------- ---------- ---------- ----------
Parse        0      0.00       0.00          0          0          0          0
```

```
Execute      4      0.00       0.00          0          0          0          0
Fetch        4      0.00       0.01          5         12          0          4
------- ------ -------- ---------- ---------- ---------- ---------- ----------
total        8      0.00       0.01          5         12          0          4

Misses in library cache during parse: 0
Optimizer mode: RULE
Parsing user id: SYS  (recursive depth: 2)
```

本例中由于表来源的信息没有记录在跟踪文件里，并且用户CHRIS没有这条SQL语句涉及的对象权限，因此没有显示执行计划（参见第10章关于执行EXPLAIN PLAN语句所需权限的详细信息）。输出部分接下来是等待事件。

```
Elapsed times include waiting on following events:
  Event waited on                             Times Waited  Max. Wait Total Waited
  ----------------------------------------  ------------  ---------- ------------
  db file sequential read                              5        0.00         0.01
```

在所有SQL语句的报告后，你可以看到执行统计信息、解析和等待事件的综合统计。这里唯一需要注意的就是非递归SQL从递归SQL里分离出来。

```
OVERALL TOTALS FOR ALL NON-RECURSIVE STATEMENTS

call     count       cpu    elapsed       disk      query    current       rows
------- ------ -------- ---------- ---------- ---------- ---------- ----------
Parse        2      0.00       0.00          0          0          0          0
Execute      3      0.45       0.42         20        226          0          3
Fetch        0      0.00       0.00          0          0          0          0
------- ------ -------- ---------- ---------- ---------- ---------- ----------
total        5      0.45       0.42         20        226          0          3

Misses in library cache during parse: 2
Misses in library cache during execute: 1

Elapsed times include waiting on following events:
  Event waited on                             Times Waited  Max. Wait Total Waited
  ----------------------------------------  ------------  ---------- ------------
  SQL*Net message to client                            2        0.00         0.00
  SQL*Net message from client                          2        0.00         0.00

OVERALL TOTALS FOR ALL RECURSIVE STATEMENTS

call     count       cpu    elapsed       disk      query    current       rows
------- ------ -------- ---------- ---------- ---------- ---------- ----------
Parse        2      0.00       0.00          0          0          0          0
Execute     29      0.00       0.00          0          0          0          0
Fetch    10037      6.50      11.97      71569      38832          8    1000028
------- ------ -------- ---------- ---------- ---------- ---------- ----------
Total    10068      6.50      11.97      71569      38832          8    1000028

Misses in library cache during parse: 2
Misses in library cache during execute: 2

Elapsed times include waiting on following events:
```

```
Event waited on                             Times Waited  Max. Wait Total Waited
----------------------------------------  ------------  ---------- ------------
db file sequential read                              72        0.00         0.04
db file scattered read                              530        0.06         2.79
direct path write temp                            11002        0.00         0.51
direct path read temp                             24015        0.00         2.41
```

接下来的几行是对当前会话的SQL数量进行概括，包括由数据库引擎递归执行的数量和EXPLIAN PLAN执行的次数：

```
 5  user SQL statements in session.
13  internal SQL statements in session.
18  SQL statements in session.
 2  statements EXPLAINed in this session.
```

现在，输出文件已经包含了跟踪文件的所有信息。首先，你能看到跟踪文件名、其版本和用于分析的参数sort的值。接着是所有会话数和SQL语句。在这个例子中，跟踪文件里少了14（18–4）行SQL语句是因为指定了参数print=4。同时也记录了执行EXPLAIN PLAN的表信息。最后，是所有SQL语句的总运行时间（以秒为单位）。我个人更愿意在跟踪文件的头部看到最后这部分信息，因为每次我打开TKPROF的输出文件时，总是最先浏览最后部分。关键是要知道整个跟踪文件花费多少时间，否则你无法判断一个SQL语句对于总响应时间的影响程度。

```
Trace file: DBM11203_ora_28030.trc
Trace file compatibility: 11.1.0.7
Sort options: prsela    exeela    fchela
       1  session in tracefile.
       5  user SQL statements in trace file.
      13  internal SQL statements in trace file.
      18  SQL statements in trace file.
      18  unique SQL statements in trace file.
       2  SQL statements EXPLAINed using schema:
           CHRIS.prof$plan_table
             Default table was used.
             Table was created.
             Table was dropped.
   46125  lines in trace file.
      12  elapsed seconds in trace file.
```

3.1.6　使用 TVD$XTAT

Trivadis Extended Tracefile Analysis Tool（TVD$XTAT）是命令行工具。与TKPROF一样，TVD$XTAT的主要功能是输入原始跟踪文件，生成一个格式化后的文件。输出文件可以是HTML或者文本文件。

最简单的分析是通过仅指定输入和输出文件来执行的。在下面的例子中，输入文件是DBM11106_ora_6334.trc，输出文件是DBM11106_ora_6334.html：

```
tvdxtat -i DBM11106_ora_6334.trc -o DBM11106_ora_6334.html
```

1. 为什么TKPROF不能满足需要

1999年末，通过Oracle Support文档*Interpreting Raw SQL_TRACE and DBMS_SUPPORT.START_TRACE output*（39817.1），我第一次接触到扩展SQL跟踪。从那时开始，我就明白要理解应用连接Oracle

数据库后做了什么，这些信息是必不可少的。同时，令我非常失望的是，没有工具能分析扩展的SQL跟踪文件以利用它们的内容。我注意到那个时候TKPROF还不能提供等待事件信息。利用命令行工具（比如awk）手工从原始跟踪文件里提取信息花了我大量的时间，因此我决定自己写一个分析工具，并将其命名为TVD$XTAT。

当前，TKRPFO可以提供等待事件信息，但仍存在五个主要问题，这些问题已经在TVD$XTAT中得到了解决。

- 只要指定了参数sort，那么SQL语句之间的联系就没有了。
- 数据只能以聚合的形式显示。因此，丢失了很多有用的信息。
- 不提供绑定变量的信息。
- 在TKPROF里，当SQL语句执行时间不计算在elapsed time中时，使用空闲等待事件代替（比如SQL*Net message from client）。这样做的结果就是当SQL语句根据elapsed time进行排序时，输出结果会造成误导，或者在一些极端情况下，出现大量无法解释的时间消耗。
- 当跟踪文件里没有SQL语句文本（特别是关键字PARSING IN CURSOR和END OF STMT之间的文本）时，TKPROF不会记录关于这些SQL语句的细节信息；而只是把它记录到输出文件最后资源使用率的统计中。请注意，如果开启SQL跟踪是在执行已经开始后，那么跟踪文件里就不会记录SQL语句的文本。

2. 安装

以下是安装TVD$XTAT的步骤。

(1) 从http://top.antognini.ch下载（免费软件）TVD$XTAT。

(2) 将文件解压到一个空目录。

(3) 修改用于启动TVD$XTAT的shell脚本（根据操作系统不同，要么是tvdxtat.cmd，要么是tvdxtat.sh）中的变量java_home和tvdxtat_home。前者引用的是JRE（版本1.4.2或以上版本）的安装目录。后者引用的是分发文件的解压目录。

(4) 根据需要更改命令行参数的默认值。你需要更改config目录里的tvdxtat.properties文件。可以定制默认配置，这样就不用在每次运行TVD$XTAT时指定所有参数。

(5) 也可以根据需要更改日志配置。为此，你需要更改config目录里的logging.properties文件。默认情况下，TVD$XTAT会显示错误和警告信息。通常没有必要修改它。

3. TVD$XTAT参数

如果执行TVD$XTAT时未附加任何参数，那么返回的是完整的参数列表，其中包含一个简短描述。请注意，针对每个参数，都有一个短格式（例如-c）和一个长格式（例如--cleanup）。

```
usage: tvdxtat [-c no|yes] [-f <int>] [-l <int>] [-r 7|8|9|10|11|12]
               [-s no|yes] [-t <template>] [-w no|yes]
               [-x severe|warning|info|fine|finer] -i <input> -o <output>
-c,--cleanup     remove temporary XML file (no|yes)
-f,--feedback    display progress every x lines (integer number >= 0, no
                 progress = 0)
-h,--help        display this help information and exit
-i,--input       input trace file name (valid extensions: trc|gz|zip)
-l,--limit       limit the size of lists (e.g. number of statements) in
```

```
                    the output file (integer number >= 0, unlimited = 0)
-o,--output         output file name (a temporary XML file with the same
                    name but with the extension xml is also created)
-r,--release        major release of the database engine that generated the
                    input trace file (7|8|9|10|11|12)
-s,--sys            report information about SYS recursive statements
                    (no|yes)
-t,--template       name of the XSL template used to generate the output
                    file (html|text)
-v,--version        print product version and exit
-w,--wait           report detailed information about wait events (no|yes)
-x,--logging        logging level (severe|warning|info|fine|finer)
```

各个参数的作用如下。

- input：指定输入文件。输入文件必须是包含一个或多个跟踪文件信息的跟踪文件（后缀名.trc）或压缩文件（后缀名.gz或.zip）。但要注意，只会有一个跟踪文件从.zip文件中提取出来。
- output：指定输出文件。在工具执行时会生成一个与输出文件同名但后缀名为.xml的临时XML文件。请注意，这会覆盖其他与输出文件同名的文件。
- cleanup：指定在工具执行结束后是否删除临时生成的XML文件。通常情况下，会设置为yes。此参数仅在开发阶段用于检查中间结果时才显示出其重要性。
- feedback：指定是否显示进程信息。在处理非常大的跟踪文件时，该参数可以帮助你得知当前分析的状态。该参数指定的是新消息产生时显示的行数。如果设置为0，就不会显示进程信息。
- help：指定是否显示帮助信息。该参数不能与其他参数一起使用。
- limit：设置输出文件中列表（比如，SQL语句的列表，等待和绑定变量的列表）的最大行数。如果设置成0，那么就没有限制。
- release：指定生成输入跟踪文件的数据库主要版本（即7、8、9、10、11或12）。
- sys：指定输出文件中是否显示由sys用户执行的递归SQL语句信息。通常设置成no。
- template：指定生成输出文件的XSL模板名。默认情况下，有两个模板可用：html.xsl和text.xsl。前者生成HTML输出文件，后者生成文本输出文件。你可以修改默认模板，也可以创建新模板。这样就可以完全定制输出文件。模板必须要放在templates文件夹中。
- version：指定是否显示TVD$XTAT版本号。该参数不能与其他参数一起使用。
- wait：指定是否显示等待事件的详细信息。开启此功能（即，设置参数为yes）可能会在处理时造成很大的开销。因此，建议首先设成no，当基础的等待信息无法满足分析时，再把它设置成yes重新运行一次。
- logging：控制日志级别。该参数可设置的值为：severe、warning、info、fine和finer。该参数仅在诊断工具运行时有用。

4. 解释TVD$XTAT的输出

这部分使用与TKPROF相同的跟踪文件。鉴于TVD$XTAT的输出结构是根据TKPROF设计的，这里我只会介绍TVD$XTAT特有的部分。我使用如下命令来生成输出文件：

```
tvdxtat -i DBM11203_ora_28030.trc -o DBM11203_ora_28030.txt -s no -w yes -t text
```

注意，跟踪文件以及输出的HTML文件都随本章的其他文件一起可供下载。

输出文件最开始是输入跟踪文件的汇总信息。其中最重要的是跟踪文件里的interval和transaction数。

```
Database Version
****************
Oracle Database 11g Enterprise Edition Release 11.2.0.3.0 - 64bit Production
With the Partitioning, Automatic Storage Management, Oracle Label Security, OLAP,
Data Mining and Real Application Testing options

Analyzed Trace File
*******************
/u00/app/oracle/diag/rdbms/dbm11203/DBM11203/trace/DBM11203_ora_28030.trc

Interval
********
Beginning 30 Nov 2012 23:21:45.691
End       30 Nov 2012 23:21:58.097
Duration  12.407 [s]

Transactions
************
Committed 0
Rollbacked 0
```

分析输出文件要从汇总资源使用分析开始。这里的处理用了12.407秒。大约56%的时间用在了CPU执行上，大约24%用于读写临时文件（direct path read temp和direct path write temp），23%用于读取数据文件（db file scattered read和db file sequential read）。总之就是大部分时间用在了CPU执行上，剩下的时间用来处理磁盘I/O。请注意，未被计算的时间是明确给出的。

```
Resource Usage Profile
**********************

                                   Total            Number of Duration per
Component                     Duration [s]       %     Events   Events [s]
--------------------------- ------------ ------- --------- ------------
CPU                                6.969  56.171       n/a          n/a
db file scattered read             2.792  22.502       530        0.005
direct path read temp              2.417  19.479    24,015        0.000
direct path write temp             0.513   4.136    11,002        0.000
db file sequential read            0.041   0.326        72        0.001
SQL*Net message from client        0.001   0.008         2        0.001
SQL*Net message to client          0.000   0.000         2        0.000
unaccounted-for                   -0.325  -2.623       n/a          n/a
--------------------------- ------------ -------
Total                             12.407 100.000
```

注意 TVD$XTAT会根据响应时间对列表进行排序。没有选项可改变这一行为，因为只有这种排序才对研究性能问题有意义。

通过大致的描述只能知道数据库引擎花费的时间。为了继续分析，就需要找出哪些SQL语句消耗了大量的时间。为了达到这个目的，在汇总资源使用分析之后是一个包含所有非递归SQL语句的列表。

这样，你就能知道单独一个SQL语句（实际上是PL/SQL块）在整个执行时间中占用的比例。请注意下面的列表中总和并不是100%，因为未被计算的时间被省略了。

```
The input file contains 18 distinct statements, 15 of which are recursive.
In the following table, only non-recursive statements are reported.

                            Total               Number of Duration per
Statement ID Type    Duration [s]         % Executions Execution [s]
------------ ------ ------------ ------- ---------- -------------
#1           PL/SQL       12.724 102.561          1        12.724
#5           PL/SQL        0.006   0.045          1         0.006
#9           PL/SQL        0.002   0.016          1         0.002
------------ ------ ------------ -------
Total                     12.732 102.623
```

下一步通常来说应该找到占用最多执行时间的SQL信息。为了能更容易找到SQL，TVD$XTAT为每个SQL生成了一个标识符（上面列表里的Statement ID列）。在HTML类型的输出文件中，你可以单击对应标识符来找到SQL语句的详细信息。在文本类型的输出文件中，你必须手动搜索字符串“STATEMENT #1”。

接下来的信息来自每条SQL语句：执行环境的基本信息、SQL语句的执行统计信息、执行计划、执行用到的绑定变量和等待事件。只有在跟踪文件里记录了执行计划，绑定变量和等待事件的信息才会在这里显示。

首先，是执行环境的基本信息和SQL语句文本。请注意，会话属性信息仅在设置之后才会显示。比如，这里的属性动作名因为应用没有设置而不会显示。另外需要注意的是，从11.1版本开始，生成跟踪文件才能使用SQL ID。

```
Session ID                 156.29
Service Name               SYS$USERS
Module Name                SQL*Plus
Parsing User               34
Hash Value                 166910891
SQL ID                     15p0p084z5qxb

DECLARE
  l_count INTEGER;
BEGIN
  FOR c IN (SELECT extract(YEAR FROM d), id, pad
            FROM t
            ORDER BY extract(YEAR FROM d), id)
  LOOP
    NULL;
  END LOOP;
  FOR i IN 1..10
  LOOP
    SELECT count(n) INTO l_count
    FROM t
    WHERE id < i*123;
  END LOOP;
END;
```

执行统计信息会根据数据库调用类型聚合并以表格形式显示。由于表结构是根据TKPROF生成的，因此各列意义相同。但这里又新增了两列：Misses和LIO。前者是每类调用期间的硬解析数。后者是列Consistent和Current的总和。同时请注意TVD$XTAT提供了两张表。第一张表包含所有与当前语句相关的递归SQL统计信息。第二张表与TKPROF一样，不包含统计信息。

```
Database Call Statistics with Recursive Statements
****************************************************

Call    Count Misses   CPU Elapsed    PIO    LIO  Consistent  Current  Rows
------- ----- ------ ----- ------- ------ ------  ----------  -------  ----
Parse       1      1 0.005   0.006      7     20          20        0     0
Execute     1      0 6.957  12.387 71,562 38,820      38,812        8     1
Fetch       0      0 0.000   0.000      0      0           0        0     0
------- ----- ------ ----- ------- ------ ------  ----------  -------  ----
Total       2      1 6.962  12.393 71,569 38,840      38,832        8     1

Database Call Statistics without Recursive Statements
*******************************************************
Cal     Count Misses   CPU Elapsed    PIO    LIO Consistent Current Rows
------- ----- ------ ----- ------- ------ ------ ---------- ------- ----
Parse       1      1 0.005   0.004      0      0          0       0    0
Execute     1      0 0.448   0.410      0      0          0       0    1
Fetch       0      0 0.000   0.000      0      0          0       0    0
------- ----- ------ ----- ------- ------ ------ ---------- ------- ----
Total       2      1 0.453   0.414      0      0          0       0    1
```

在这里，根据执行统计信息可发现当前SQL语句几乎没有花费时间。这点同样在下面的资源使用分析中得以体现。实际上，这里显示大约96%的时间用在了递归SQL语句上。

```
Component                  Duration [s]       % Events Events [s]
-------------------------- ------------ ------- ------ ----------
recursive statements             12.271  96.437    n/a        n/a
CPU                               0.453   3.560    n/a        n/a
SQL*Net message from client       0.000   0.003      1      0.000
SQL*Net message to client         0.000   0.000      1      0.000
-------------------------- ------------ -------
Total                            12.724 100.000
```

在资源使用分析之后列出的都是那些递归SQL语句。你可以看到statement 2的SQL语句，这是一个SELECT查询，占用了96%的响应时间。请注意，这里除了statement 3的SQL语句以外，其他都是由数据库引擎（比如，在解析阶段）产生的，因此带有SYS recursive标签。

```
7 recursive statements were executed.

                                              Total
Statement ID Type                      Duration [s]       %
------------ --------------------- ------------ -------
#2           SELECT                      12.234  96.150
#3           SELECT                       0.033   0.263
#7           SELECT (SYS recursive)       0.003   0.022
#11          SELECT (SYS recursive)       0.000   0.001
#12          SELECT (SYS recursive)       0.000   0.001
#14          SELECT (SYS recursive)       0.000   0.001
#16          SELECT (SYS recursive)       0.000   0.000
```

```
------------ ---------------------- ------------ -------
Total                                      12.252  96.286
```

由于statement 2的SQL占用了最多的响应时间，因此你需要继续深入并获得更多的详细信息。它的结构基本上与statement 1 SQL一样，但是还包括附加信息。在显示执行环境的部分，你可以看到递归级别（记住，应用是在级别0执行得SQL）和父SQL的statement编号。后者可以保证不会丢失SQL语句之间的联系（TKPROF就没有这种联系）。

```
Session ID           156.29
Service Name         SYS$USERS
Module Name          SQL*Plus
Parsing User         34
Recursive Level      1
Parent Statement ID  1
Hash Value           955957303
SQL ID               7wd0gdwwgph1r

SELECT EXTRACT(YEAR FROM D), ID, PAD FROM T ORDER BY EXTRACT(YEAR FROM D), ID
```

下一步，如果跟踪文件里有执行计划，那么你应该能找到。它的格式与TKRPOF生成的输出相同：

```
Execution Plan
**************

Optimizer Mode       ALL_ROWS
Hash Value           961378228

     Rows Operation
--------- -------------------------------------------------------------
1,000,000 SORT ORDER BY (cr=38474 pr=71499 pw=33035 time=11123996 us
                         cost=216750 size=264000000 card=1000000)
1,000,000  TABLE ACCESS FULL T (cr=38474 pr=38463 pw=0 time=5674541 us
                                cost=21 size=264000000 card=1000000)
```

通常对于所有SQL语句都是一样的，执行计划后面是执行统计信息、资源使用分析，有可能也有级别2的递归SQL语句（你当前看到的是级别1的SQL语句）。在本例中，递归SQL只占了不到1%的响应时间。换句话说，statement 2的SQL语句占用了所有的响应时间：

```
Database Call Statistics with Recursive Statements
**************************************************

Call                 Count Misses   CPU Elapsed    PIO    LIO Consistent Current      Rows
----------------- ------ ------ ----- ------- ------ ------ ---------- ------- ---------
Parse                    1      1 0.004   0.010      7     32         32       0         0
Execute                  1      1 0.000   0.000      0      0          0       0         0
Fetch               10,001      0 6.492  11.926 71,499 38,482     38,474       8 1,000,000
----------------- ------ ------ ----- ------- ------ ------ ---------- ------- ---------
Total               10,003      2 6.496  11.936 71,506 38,514     38,506       8 1,000,000
Average (per row)        0      0 0.000   0.000      0      0          0       0         1

Database Call Statistics without Recursive Statements
*****************************************************

Call                 Count Misses   CPU Elapsed    PIO    LIO Consistent Current      Rows
```

```
----------------- ------ ------ ----- ------- ------ ------ ---------- ------- ---------
Parse                  1      1 0.001  0.001      0      9          9       0         0
Execute                1      1 0.000  0.000      0      0          0       0         0
Fetch             10,001      0 6.492 11.926 71,499 38,482     38,474       8 1,000,000
----------------- ------ ------ ----- ------- ------ ------ ---------- ------- ---------
Total             10,003      2 6.493 11.927 71,499 38,491     38,483       8 1,000,000
Average (per row)      0      0 0.000  0.000      0      0          0       0         1
```



```
Resource Usage Profile
**********************

                              Total          Number of Duration per
Component               Duration [s]       %    Events   Events [s]
----------------------- ------------ ------- --------- ------------
CPU                            6.493  53.071       n/a          n/a
db file scattered read         2.792  22.818       530        0.005
direct path read temp          2.417  19.753    24,015        0.000
direct path write temp         0.513   4.194    11,002        0.000
recursive statements           0.020   0.161       n/a          n/a
db file sequential read        0.000   0.002         2        0.000
----------------------- ------------ -------
Total                         12.234 100.000

6 recursive statements were executed.

                                                Total
Statement ID Type                        Duration [s]       %
------------ --------------------- ------------ -------
#4           SELECT (SYS recursive)        0.015   0.121
#6           SELECT (SYS recursive)        0.004   0.032
#10          SELECT (SYS recursive)        0.001   0.008
#13          SELECT (SYS recursive)        0.000   0.001
#17          SELECT (SYS recursive)        0.000   0.000
#18          SELECT (SYS recursive)        0.000   0.000
------------ --------------------- ------------ -------
Total                                      0.006   0.050
```

在以上的资源使用分析里，等待事件被分组汇总显示。为了获得更多信息，以下显示的是针对资源使用分析里每部分的直方图。在本例中，显示的信息与statement 2 SQL的等待事件db file scattered read相关。请注意，等待事件根据区间（列Range）进行分组。例如，你可以看到52%的等待事件持续了4096~8192微秒。由于多块读的等待事件是db file scattered read，查看磁盘I/O操作（列Blocks per Event）读取的平均块数也能获取有用的信息。

```
                                Total           Number of            Duration per          Blocks per
Range [µs]                   Duration        %     Events        %     Event [µs] Blocks       Event
------------------------ -------- ------- --------- ------- ------------ ------ ----------
256 ≤ duration < 512        0.003   0.111         7       1          443     56          8
512 ≤ duration < 1024       0.008   0.288         9       2          892     72          8
1024 ≤ duration < 2048      0.033   1.191        18       3        1,847    826         46
2048 ≤ duration < 4096      0.517  18.525       166      31        3,115 11,627         70
4096 ≤ duration < 8192      1.465  52.459       264      50        5,547 20,742         79
8192 ≤ duration < 16384     0.579  20.736        60      11        9,648  4,722         79
16384 ≤ duration < 32768    0.126   4.496         5       1       25,101    336         67
```

```
32768 ≤ duration < 65536     0.061   2.195           1       0       61,274     81          81
------------------------ -------- ------- --------- ------- ------------ ------ ----------
Total                        2.792 100.000         530 100.000        5,267 38,462          73
```

如果开启了详细信息显示（使用参数wait），之前的直方图可能会显示更多的细节。但这与等待事件类型关系很大。实际上，许多事件没有附加信息。磁盘I/O操作相关的等待事件也只是显示文件级别的信息。例如，以下表格显示statement 2 SQL的等待事件db file scattered read的信息。在这个例子中，在2.792秒时间里磁盘I/O在data file 4上执行了530次操作。这代表平均每次磁盘I/O操作持续了5.267毫秒（注意此表单位：毫秒）。

```
File          Total          Number of                                Duration per
Number   Duration [s]      %    Events        %   Blocks [b]       %   Event [µs]
------- ------------ ------- ---------- ------- ----------- ------- -------------
4              2.792 100.000        530 100.000      38,462 100.000         5,267
```

与你预料的一样，所有SQL语句的结构都是相同的。但是有一块信息在前两条SQL语句里不显示。让我们引用一部分使用绑定变量的SQL语句的信息来举例。正如输出statement 3 SQL显示的那样，如果绑定变量信息已经记录在跟踪文件里，那么TVD$XTAT会显示它们的数据类型和值。另外，如果存在多次执行（本例中是10次），绑定变量会用执行次数标记。请看下面的例子。

```
Session ID          156.29
Service Name        SYS$USERS
Module Name         SQL*Plus
Parsing User        34
Recursive Level     1
Parent Statement ID 1
Hash Value          1035370675
SQL ID              7fjjjf0yvd05m

SELECT COUNT(N) FROM T WHERE ID < :B1 *123

Bind Variables
**************

10 bind variable sets were used to execute this statement.

Number of
Execution  Bind  Datatype  Value
---------- ----- --------- ------
1          1     NUMBER    "1"
2          1     NUMBER    "2"
3          1     NUMBER    "3"
4          1     NUMBER    "4"
5          1     NUMBER    "5"
6          1     NUMBER    "6"
7          1     NUMBER    "7"
8          1     NUMBER    "8"
9          1     NUMBER    "9"
10         1     NUMBER    "10"
```

总之，虽然执行了大量SQL语句（一共是18条），statement 2的SQL占用了大多数的响应时间。因此，为了提高性能，这条SQL应该需要优化或者删除。

3.2 探查 PL/SQL 代码

数据库引擎提供了两个在PL/SQL引擎中集成的探查器，以便在探查PL/SQL代码时使用。一个是通过包dbms_profiler进行管理的行级别探查器。另外一个是通过包dbms_hprof进行管理的调用级别探查器（也称为分层探查器，hierarchical profiler）。表3-2汇总了每种探查器的主要优势。

表3-2 DBMS_HPROF和DBMS_PROFILER的主要优势

DBMS_HPROF	DBMS_PROFILER
开启后对开销影响非常小	提供行级别的信息
提供调用级别的信息	11.1版本之前就可以使用
有“self time”和“total time”的概念	所有主要开发工具都支持
并不需要附加的权限	
支持native-compiled PL/SQL	

分层探查器探查器提供的运行时统计信息不仅要比行级别探查器更精确，同时也更有用。除非你确实需要行级别的信息。因此，如果可能，建议使用分层探查器，除非你有特别的需求，需要使用行级别提供的信息。

3.2.1 使用 DMBS_HPROF

借助11.1版本中引入的包dbms_hprof，你可以在会话级别启用和禁用分层探查器。启用之后，会为执行的每个PL/SQL和SQL调用收集以下信息：

- 调用执行的总次数；
- 处理调用花费的时间；
- 处理子调用花费的时间；
- 调用层次结构信息。

用户在会话级别可以执行（有一个限制是，封装的PL/SQL代码只允许你收集顶级调用的信息）的所有PL/SQL代码（比如在包和触发器中的代码）都会被收集。你只需要对包dbms_hprof有EXECUTE权限就可以启用探查。

探查期间收集到的数据会存储到操作系统级别上的某个跟踪文件中。然后，为了探查目的，可以将该数据像图3-4所示的那样加载到数据库表中，或者也可以使用PLSHPROF实用工具对其进行处理。

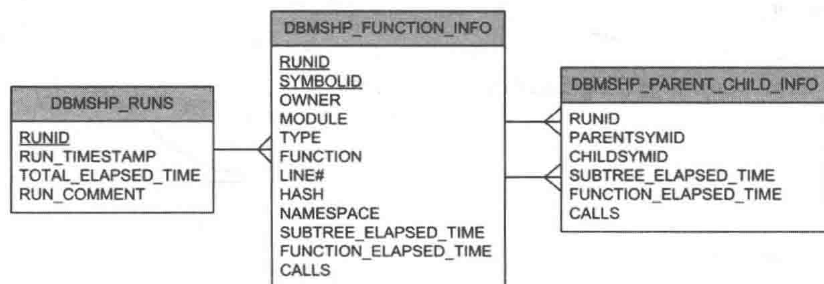


图3-4 探查器将收集到的信息存储到三个数据库表中，请注意带下划线的字段为主键

关于已探查会话的信息会保存在表dbmshp_runs中。为每次运行执行的子程序列表保存在表dbmshp_function_info中。调用方与被调用方之间的父子关系保存在表dbmshp_parent_child_info中。换句话说，表dbmshp_parent_child_info包含用于重构调用层次结构的信息。

1. 创建输出表

包dbms_hprof是以执行它的用户的权限来运行的。因此，输出表并不需要由sys用户创建。要么由数据库管理员安装一次输出表（通过运行dbmshptab.sql脚本），并提供使用这些表的必要同义词和权限，要么由每个用户在自己的schema下安装。在下面这个例子中，由数据库管理员安装一次输出表：

```
CONNECT / AS SYSDBA
@?/rdbms/admin/dbmshptab.sql

CREATE PUBLIC SYNONYM dbmshp_runs FOR dbmshp_runs;
CREATE PUBLIC SYNONYM dbmshp_function_info FOR dbmshp_function_info;
CREATE PUBLIC SYNONYM dbmshp_parent_child_info FOR dbmshp_parent_child_info;
CREATE PUBLIC SYNONYM dbmshp_runnumber FOR dbmshp_runnumber;

GRANT SELECT, INSERT, UPDATE, DELETE ON dbmshp_runs TO PUBLIC;
GRANT SELECT, INSERT, UPDATE, DELETE ON dbmshp_function_info TO PUBLIC;
GRANT SELECT, INSERT, UPDATE, DELETE ON dbmshp_parent_child_info TO PUBLIC;
GRANT SELECT ON dbmshp_runnumber TO PUBLIC;
```

2. 收集探查数据

通过调用start_profiling过程来启用探查器，开始探查分析。该过程支持三个参数。

- location：指定包含探查数据的跟踪文件的存放位置，需要指定操作系统级别的目录名。
- filename：指定跟踪文件名。如果文件存在，会直接覆盖。
- max_depth：指定探查数据收集是否受到指定调用深度限制。默认情况下（NULL），没有限制。

启用探查器之后，会收集由PL/SQL引擎执行的代码探查数据。调用stop_profiling过程可禁用探查。

包含探查数据的跟踪文件可用之后，就立即可以通过调用analyze函数将跟踪文件加载到输出表中。调用analyze函数需要指定两个参数：location和filename。这两个参数的作用与start_profiling过程中的同名参数一模一样。因此，你应该把它们设置成与start_profiling过程相同的值。analyze函数也支持其他参数，你可以在*PL/SQL Packages and Types Reference*手册中查看这些参数。

下面的例子引自脚本dbms_hprof.sql生成的输出。这个例子为了探查一个匿名PL/SQL块而做了一次最小限度的执行。将探查器数据加载到数据库中时所选择的runid值会在下一部分对探查会话的输出进行分析时使用到。

```
SQL> BEGIN
  2    dbms_hprof.start_profiling(location => 'PLSHPROF_DIR',
  3                               filename => 'dbms_hprof.trc');
  4  END;
  5  /

SQL> DECLARE
  2    l_count INTEGER;
  3  BEGIN
  4   perfect_triangles(1000);
```

```
  5  SELECT count(*) INTO l_count
  6  FROM all_objects;
  7 END;
  8 /

SQL> BEGIN
  2   dbms_hprof.stop_profiling;
  3 END;
  4 /

SQL> SELECT dbms_hprof.analyze(location => 'PLSHPROF_DIR',
  2                            filename => 'dbms_hprof.trc') AS runid
  3 FROM dual;

     RUNID
----------
         1
```

一旦将探查数据加载到输出表中，就该生成报表了。下面几节会介绍用来生成报表的三种主要方法。

3. 手动生成探查数据报表

如本节所示，将探查数据存入输出表后，就可以进行正常的查询。下面的例子引自dbms_hprof.sql脚本生成的输出。

第一个查询把探查数据按照命名空间进行分组。在本例中，你可以看到PL/SQL代码占用的响应时间比例（这里是45.1%），你只有继续在探查器提供的数据里找到更多的细节才能找出这代表的意义。另一方面，如果你发现SQL占用了大多数的响应时间，那么使用PL/SQL探查器就是错误的；你可以使用SQL跟踪等更好的工具来找出哪些SQL运行缓慢。不管哪种方式，第一个查询提供了有用的信息，它能帮助你了解接下来需要关注哪些地方。

下面是第一个查询输出的例子：

```
SQL> SELECT sum(function_elapsed_time)/1000 AS total_ms,
  2         100*ratio_to_report(sum(function_elapsed_time)) over () AS total_percent,
  3         sum(calls) AS calls,
  4         100*ratio_to_report(sum(calls)) over () AS calls_percent,
  5         namespace AS namespace_name
  6 FROM dbmshp_function_info
  7 WHERE runid = 1
  8 GROUP BY namespace
  9 ORDER BY total_ms DESC;

TOTAL [ms]   TOT%       CALLS   CAL% NAMESPACE_NAME
---------- ------ ---------- ------ ---------------
       565   54.9         89    5.6 SQL
       464   45.1      1,494   94.4 PLSQL
```

第二个查询与第一个查询很像，它把探查数据按照模块级别分组。在这里，可以看到perfect_triangles过程占用了PL/SQL的大部分响应时间（44.9%）。

```
SQL> SELECT sum(function_elapsed_time)/1000 AS total_ms,
  2        100*ratio_to_report(sum(function_elapsed_time)) over () AS total_percent,
  3        sum(calls) AS calls,
```

```
  4        100*ratio_to_report(sum(calls)) over () AS calls_percent,
  5        namespace,
  6        nvl(nullif(owner || '.' || module, '.'), function) AS module_name,
  7        type
  8  FROM dbmshp_function_info
  9  WHERE runid = 1
 10  GROUP BY namespace, nvl(nullif(owner || '.' || module, '.'), function), type
 11  ORDER BY total_ms DESC;

TOTAL [ms]  TOT%  CALLS CAL% NAMESPACE MODULE_NAME                 TYPE
---------- ----- ------ ----- --------- --------------------------- ------------
       521  50.6      1   0.1 SQL       __static_sql_exec_line5
       462  44.9  1,214  76.7 PLSQL     CHRIS.PERFECT_TRIANGLES     PROCEDURE
        44   4.3     88   5.6 SQL       SYS.XML_SCHEMA_NAME_PRESENT PACKAGE BODY
         1   0.1     44   2.8 PLSQL     SYS.XML_SCHEMA_NAME_PRESENT PACKAGE BODY
         1   0.1      3   0.2 PLSQL     __plsql_vm
         0   0.0      3   0.2 PLSQL     __anonymous_block
         0   0.0     46   2.9 PLSQL     __plsql_vm@1
         0   0.0    179  11.3 PLSQL     SYS.DBMS_OUTPUT             PACKAGE BODY
         0   0.0      1   0.1 PLSQL     SYS.DBMS_UTILITY            PACKAGE BODY
         0   0.0      1   0.1 PLSQL     SYS.DBMS_SESSION            PACKAGE BODY
         0   0.0      1   0.1 PLSQL     SYS.DBMS_APPLICATION_INFO   PACKAGE BODY
         0   0.0      1   0.1 PLSQL     SYS.DBMS_APPLICATION_INFO   PACKAGE SPEC
         0   0.0      1   0.1 PLSQL     SYS.DBMS_HPROF              PACKAGE BODY
```

第三个查询的目的是分层并在一个更好的级别进行分组（包括所有的PL/SQL调用），它并不只是简单地显示调用分层，同时也会显示调用方和被调用方花费的时间。例如，你可以看到调用perfect_triangles花费了463毫秒，该过程本身花费了393毫秒。被调用方sides_are_unique和store_dup_sides占用了剩下的69毫秒（64+5），它们并没有出现在之前的查询中。

```
SQL> SELECT lpad(' ', (level-1) * 2) || nullif(c.owner || '.', '.') ||
  2          CASE WHEN c.module = c.function
  3               THEN c.function
  4               ELSE nullif(c.module || '.', '.') || c.function END AS function_name,
  5          pc.subtree_elapsed_time/1000 AS total_ms,
  6          pc.function_elapsed_time/1000 AS function_ms,
  7          pc.calls AS calls
  8  FROM dbmshp_parent_child_info pc,
  9      dbmshp_function_info p,
 10      dbmshp_function_info c
 11  START WITH pc.runid = 1
 12  AND p.runid = pc.runid
 13  AND c.runid = pc.runid
 14  AND pc.childsymid = c.symbolid
 15  AND pc.parentsymid = p.symbolid
 16  AND p.symbolid = 1
 17  CONNECT BY pc.runid = prior pc.runid
 18  AND p.runid = pc.runid
 19  AND c.runid = pc.runid
 20  AND pc.childsymid = c.symbolid
 21  AND pc.parentsymid = p.symbolid
 22  AND prior pc.childsymid = pc.parentsymid
 23  ORDER SIBLINGS BY total_ms DESC;
```

```
FUNCTION NAME                                              TOTAL [ms] FUNCTION [ms]  CALLS
---------------------------------------------------------- ---------- ------------- ------
__static_sql_exec_line5                                           566           521      1
  __plsql_vm@1                                                     45             0     46
    SYS.XML_SCHEMA_NAME_PRESENT.IS_SCHEMA_PRESENT                  45             1     44
      SYS.XML_SCHEMA_NAME_PRESENT.__dyn_sql_exec_line34            22            22     44
      SYS.XML_SCHEMA_NAME_PRESENT.__dyn_sql_exec_line17            22            22     44
CHRIS.PERFECT_TRIANGLES                                           463           393      1
  CHRIS.PERFECT_TRIANGLES.PERFECT_TRIANGLES.SIDES_ARE_UNIQUE       64            64  1,034
  CHRIS.PERFECT_TRIANGLES.PERFECT_TRIANGLES.STORE_DUP_SIDES         5             5    179
  SYS.DBMS_OUTPUT.PUT_LINE                                          0             0    179
SYS.DBMS_SESSION.IS_ROLE_ENABLED                                    0             0      1
  SYS.DBMS_UTILITY.CANONICALIZE                                     0             0      1
SYS.DBMS_APPLICATION_INFO.SET_MODULE                                0             0      1
SYS.DBMS_APPLICATION_INFO.__pkg_init                                0             0      1
SYS.DBMS_HPROF.STOP_PROFILING                                       0             0
```

4. 使用PLSHPROF

可以使用命令行工具PLSHPROF来处理由dbms_hprof生成的跟踪文件。在工具执行期间会生成一些HTMP报告。如果不加任何参数执行PLSHPROF，那么返回的是PLSHPROF的完整参数列表，以及各参数的简短描述。

```
Usage: plshprof [<option>...] <tracefile1> [<tracefile2>]
  Options:
    -trace <symbol>    (no default)     specify function name of tree root
    -skip <count>      (default=0)      skip first <count> invokations
    -collect <count>   (default=1)      collect info for <count> invokations
    -output <filename> (default=<symbol>.html or <tracefile1>.html)
    -summary                            print time only
```

如你所见，可以指定一个或两个跟踪文件和多个选项。如果只指定了单独的一个跟踪文件，PLSHPROF会生成如下报告：

- 根据8个不同的条件进行分类的函数已用时间数据；
- 根据3个不同的条件进行分类的模块已用时间数据；
- 根据3个不同的条件进行分类的命名空间已用时间数据；
- 父级别与子级别的已用时间数据。

例如，以下命令处理跟踪文件dbms_hprof.trc并且生成包含一组报告的文件dbms_hprof.html：

```
plshprof -output dbms_hprof dbms_hprof.trc
```

请注意，跟踪文件和HTML报告都可以在dbms_hprof.zip中找到。图3-5显示了其中一个报告。

当你想对比同一个程序里的两个运行结果时，可以指定两个跟踪文件。例如，可以指定两个跟踪文件并且对比代码的改变对性能产生的影响。如果这两个跟踪文件不相同，那么PLSHPROF会生成一个报表集合，这些报表与为单个跟踪文件生成的报表相似，但PLSHPROF会标明两次运行之间的增量。

Module	Ind%	Cum%	Calls	Ind%	Module Name
522033	50.7%	50.7%	53	3.3%	
462495	44.9%	95.7%	1214	76.7%	CHRIS.PERFECT_TRIANGLES
44683	4.3%	100%	132	8.3%	SYS.XML_SCHEMA_NAME_PRESENT
25	0.0%	100%	179	11.3%	SYS.DBMS_OUTPUT
23	0.0%	100%	2	0.1%	SYS.DBMS_APPLICATION_INFO
22	0.0%	100%	1	0.1%	SYS.DBMS_UTILITY
21	0.0%	100%	1	0.1%	SYS.DBMS_SESSION
0	0.0%	100%	1	0.1%	SYS.DBMS_HPROF

图3-5　由PLSHPROF生成并按总函数已用时间排序的模块已用时间数据

5. 使用图形界面

除了前面提到的方法外，也可以使用第三方带有图形界面的产品。比如SQL Developer（Oracle）和Toad（Dell）。通常情况下，这些工具通过勾选复选框或单击按钮，或者直接分析输出表内容就可以探查代码了。

图3-6至图3-8显示了SQL Developer为在前几部分中阐述的探查会话提供的部分信息。

Function Calls | Module | Namespace | Call Hierarchy

Exec Time	Tot%	Calls	Cal%	Namespace
428946 µs	30.2%	1494	94.4%	PLSQL
992989 µs	69.8%	89	5.6%	SQL

图3-6　SQL Developer中显示的命名空间级别的探查数据

Function Calls | Module | Namespace | Call Hierarchy

Exec Time	Tot%	Calls	Cal%	Module
522033 µs	50.7%	53	3.3%	.
462495 µs	44.9%	1214	76.7%	CHRIS.PERFECT_TRIANGLES
44683 µs	4.3%	132	8.3%	SYS.XML_SCHEMA_NAME_PRESENT
25 µs	0.0%	179	11.3%	SYS.DBMS_OUTPUT
23 µs	0.0%	2	0.1%	SYS.DBMS_APPLICATION_INFO
22 µs	0.0%	1	0.1%	SYS.DBMS_UTILITY
21 µs	0.0%	1	0.1%	SYS.DBMS_SESSION
0 µs	0.0%	1	0.1%	SYS.DBMS_HPROF

图3-7　SQL Developer中显示的模块级别的探查数据

Function Calls	Module	Namespace	Call Hierarchy

Callee	Elapsed	Aggregated	Calls#
CHRIS.PERFECT_TRIANGLES	393062 µs	462520	1
CHRIS.PERFECT_TRIANGLES.PERFECT_TRIANGLES.SIDES_ARE_UNIQUE	64279 µs	64279	1034
CHRIS.PERFECT_TRIANGLES.PERFECT_TRIANGLES.STORE_DUP_SIDES	5154 µs	5154	179
SYS.DBMS_OUTPUT.PUT_LINE	25 µs	25	179
SYS.DBMS_APPLICATION_INFO.SET_MODULE	20 µs	20	1
SYS.DBMS_APPLICATION_INFO.__pkg_init	3 µs	3	1
SYS.DBMS_HPROF.STOP_PROFILING	0 µs	0	1
SYS.DBMS_SESSION.IS_ROLE_ENABLED	21 µs	43	1
SYS.DBMS_UTILITY.CANONICALIZE	22 µs	22	1
..__static_sql_exec_line5	520995 µs	565797	1
..__plsql_vm@1	119 µs	44802	46
SYS.XML_SCHEMA_NAME_PRESENT.IS_SCHEMA_PRESENT	750 µs	44683	44
SYS.XML_SCHEMA_NAME_PRESENT.__dyn_sql_exec_line17	21855 µs	21855	44
SYS.XML_SCHEMA_NAME_PRESENT.__dyn_sql_exec_line34	22078 µs	22078	44

图3-8 SQL Developer中显示的调用层次结构的探查数据

3.2.2 使用 DBMS_PROFILER

使用包dbms_profiler可以在会话级别启用和禁用行级别探查器。启用探查器之后，针对每行执行过的代码，会收集如下信息：

- 总执行次数；
- 执行时的总花费时间；
- 执行时花费的最短时间和最长时间。

只要用户拥有CREATE权限，那么他在会话级别执行的所有PL/SQL代码都会被收集，但不包括封装和本地编译的代码。换句话说，拥有执行一段PL/SQL代码的权限并不足以使用探查器。因此，实际上只有被探查对象的所有者或拥有CREATE ANY权限的用户才能进行探查。

图3-9显示了探查数据存储在数据库中的表结构。探查分析的信息存在表plsql_profiler_runs中。每次执行的单位列表保存在表plsql_profiler_units中。每行代码执行后的探查数据保存在表plsql_profiler_data中。

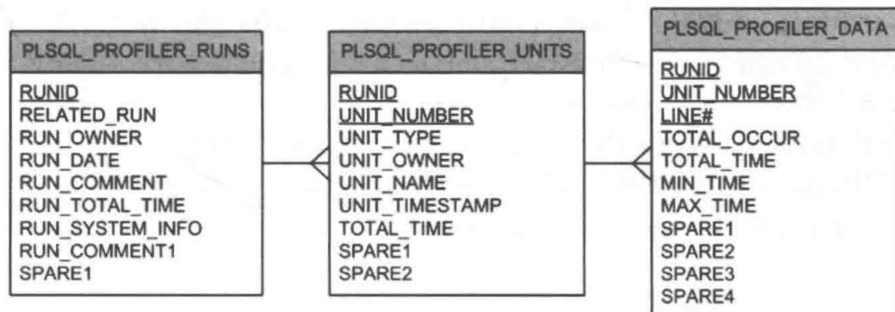


图3-9 探查器将收集到的信息存储在三个数据库表中，请注意，带下划线的字段为主键

1. 安装输出表

包是以执行它的用户的权限运行的。因此，输出表并不需要由sys用户来创建。要么由数据库管理员来安装一次输出表（通过运行dbmshptab.sql脚本），并提供使用这些输出表的必要同义词和权限，要么由每个用户在自己的schema下安装输出表。

```
CONNECT / AS SYSDBA
@?/rdbms/admin/proftab.sql

CREATE PUBLIC SYNONYM plsql_profiler_runs FOR plsql_profiler_runs;
CREATE PUBLIC SYNONYM plsql_profiler_units FOR plsql_profiler_units;
CREATE PUBLIC SYNONYM plsql_profiler_data FOR plsql_profiler_data;
CREATE PUBLIC SYNONYM plsql_profiler_runnumber FOR plsql_profiler_runnumber;

GRANT SELECT, INSERT, UPDATE, DELETE ON plsql_profiler_runs TO PUBLIC;
GRANT SELECT, INSERT, UPDATE, DELETE ON plsql_profiler_units TO PUBLIC;
GRANT SELECT, INSERT, UPDATE, DELETE ON plsql_profiler_data TO PUBLIC;
GRANT SELECT ON plsql_profiler_runnumber TO PUBLIC;
```

2. 收集探查数据

通过调用例程start_profiler，可以在启用探查器的情况下开始探查分析。启用探查器之后，会为PL/SQL引擎所执行的代码收集探查数据。除非通过调用flush_data例程执行显式刷新，否则虽然启用了探查器，也不会将任何探查数据存储到输出表中。通过调用stop_profiler例程，可以禁用探查器，并执行隐式刷新。此外，通过调用pause_profiler和resume_profiler例程，可以分别暂停和恢复探查器。图3-10显示了探查器的状态以及dbms_profiler中可用于触发状态更改的例程。

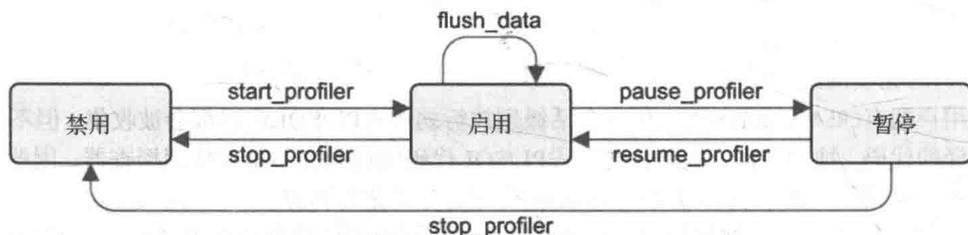


图3-10　探查器状态图。包dbms_profiler提供用于更改探查器状态（禁用、启用或暂停）的例程

针对图3-10中的每一个例程，包dbms_profiler都有对应的函数和过程。函数会返回执行结果（0=成功）。在出错时会抛出异常。除了例程start_profiler需要使用描述探查分析的两个注释作为参数，其他例程都是无参数的。

下面的例子引自脚本dbms_profiler.sql生成的输出。请注意，禁用探查器时所选择的runid值会在下一部分引用在输出表中储存的探查数据时使用到。

```
SQL> SELECT dbms_profiler.start_profiler AS status
  2  FROM dual;

    STATUS
----------
```

```
         0

SQL> execute perfect_triangles(1000)

SQL> SELECT dbms_profiler.stop_profiler AS status,
  2         plsql_profiler_runnumber.currval AS runid
  3  FROM dual;

    STATUS      RUNID
---------- ----------
         0          1
```

探查会话一结束，就应该报告由探查器生成的数据。接下来的两部分会介绍两种主要的报告方法。

3. 手工报告探查数据

由于探查数据保存在输出表中，所以可以通过正常查询来获取数据。以下是由脚本dbms_profiler.sql生成的输出。查询结果出于两个原因只提供了响应时间的百分比：首先，我们通常只关心代码最慢的部分；其次，定时信息，尤其当代码是CPU bound时，非常不可靠。实际上，对于CPU-bound处理，探查器会造成很高的开销。在本例里，就是CPU bound，处理时间从不到1秒增加到7秒左右。而在这7秒中，只有大约4秒会为探查器所用。

```
SQL> SELECT s.line,
  2         round(ratio_to_report(p.total_time) OVER ()*100,1) AS time,
  3         total_occur,
  4         s.text
  5  FROM all_source s,
  6       (SELECT u.unit_owner, u.unit_name, u.unit_type,
  7               d.line#, d.total_time, d.total_occur
  8        FROM plsql_profiler_units u, plsql_profiler_data d
  9        WHERE u.runid = 1
 10        AND d.runid = u.runid
 11        AND d.unit_number = u.unit_number) p
 12  WHERE s.owner = p.unit_owner (+)
 13  AND s.name = p.unit_name (+)
 14  AND s.type = p.unit_type (+)
 15  AND s.line = p.line# (+)
 16  AND s.owner = user
 17  AND s.name = 'PERFECT_TRIANGLES'
 18  AND s.type IN ('PROCEDURE', 'PACKAGE BODY', 'TYPE BODY')
 19  ORDER BY s.line;

LINE#     TIME%      EXEC# CODE
------ -------- ---------- ------------------------------------------------------
     1      0.0          1 PROCEDURE perfect_triangles(p_max IN INTEGER) IS
...
    29     17.7  1,105,793     FOR j IN 1..n
    30                         LOOP
    31     22.3  1,105,614       IF p_long = dup_sides(j).long
    32                              AND
    33                              p_short = dup_sides(j).short
    34                           THEN
    35      0.0        855         RETURN FALSE;
```

```
     36                              END IF;
     37                            END LOOP;
...
     44      8.2    501,500        FOR short IN 1..long
     45                            LOOP
     46     21.4    500,500          hyp := sqrt(long*long + short*short);
     47     11.0    500,500          ihyp := floor(hyp);
     48     10.5    500,500          IF hyp-ihyp < 0.01
     49                              THEN
     50      0.2     10,325            IF ihyp*ihyp = long*long + short*short
     51                                THEN
     52      0.1      1,034              IF sides_are_unique(long, short)
     53                                  THEN
     54      0.0        179                m := m+1;
     55      0.0        179                unique_sides(m).long := long;
     56      0.0        179                unique_sides(m).short := short;
     57      0.0        179                store_dup_sides(long, short);
     58                                  END IF;
     59                                END IF;
     60                              END IF;
     61                            END LOOP;
...
     69      0.0          1 END perfect_triangles;
```

Oracle针对探查数据提供了两组脚本以及查询示例。

- 如果安装了示例文件（默认情况下不安装），那么在`$ORACLE_HOME/plsql/demo/`目录下会存在脚本`profrep.sql`。
- 参见Oracle Support文档*Script to produce HTML report with top consumers out of PL/SQL Profiler DBMS_PROFILER data*（243755.1）。

4. 使用图形界面

上面介绍的手工方法也可以使用第三方工具图形界面来实现，比如PL/SQL Developer（Allround Automations）、SQLDetective（Conquest Software Solutions）、Toad和SQL Navigator（Dell）或者Rapid Sql（Embarcadero）。通常，通过在运行测试之前勾选复选框或单击按钮，或者通过直接分析输出表内容，所有这些工具都可以用于探查代码。

例如，图3-11显示了SQL Developer为前几部分中阐述的探查会话提供的信息。请注意“Total time”列中的图示，该图示高亮显示了主要的耗时代码行。

Test Window - Script for procedure PERFECT_TRIANGLES@DBM11203.ANTOGNINI.CH

Test script | DBMS Output | Statistics | Profiler | Trace

Run 21-DEC-12 Unit PERFECT_TRIANGLES - 3,963

Line	Total time	Occurrences	Text
29	702	1105793	FOR j IN 1..n
31	885	1105614	IF p_long = dup_sides(j).long
35	1	855	RETURN FALSE;
38	0	179	RETURN TRUE;
39	1	1034	END sides_are_unique;
42	327	1001	FOR long IN 1..p_max
44	326	501500	FOR short IN 1..long
46	848	500500	hyp := sqrt(long*long + short*short);
47	426	500500	ihyp := floor(hyp);
48	413	500500	IF hyp-ihyp < 0.01
50	8	10325	IF ihyp*ihyp = long*long + short*short
52	2	1034	IF sides_are_unique(long, short)
54	0	179	m := m+1;
55	0	179	unique_sides(m).long := long;
56	0	179	unique_sides(m).short := short;
57	0	179	store_dup_sides(long, short);

chris@dbm11203.antognini.ch

图3-11 PL/SQL Developer中显示的探查数据

3.2.3 触发探查器

仅可以从执行要探查的PL/SQL代码的会话内启用和禁用这两个探查器。如果探查无法手动启动，也可以像下面这样通过创建数据库触发器来为整个会话自动启用和禁用探查器：

```
CREATE TRIGGER start_hprof_profiler AFTER LOGON ON DATABASE
BEGIN
  IF (dbms_session.is_role_enabled('HPROF_PROFILE'))
  THEN
    dbms_hprof.start_profiling(
      location => 'PLSHPROF_DIR',
      filename => 'dbms_hprof_'||sys_context('userenv','sessionid')||'.trc'
    );
  END IF;
END;
/

CREATE TRIGGER stop_hprof_profiler BEFORE LOGOFF ON DATABASE
BEGIN
  IF (dbms_session.is_role_enabled('HPROF_PROFILE'))
  THEN
    dbms_hprof.stop_profiling();
  END IF;
END;
/
```

以上触发器针对的是分层探查器。可以在脚本dbms_hprof_triggers.sql和dbms_profiler_triggers.sql中找到用于为这两个探查器创建触发器的代码。如上面的触发器所示，为了避免为所有用户启用探查器，我通常建议创建一个角色（本例为hprof_profile）并且仅向测试需要的用户临时赋予权限。当然，可以只为某个单独的架构定义触发器或者设置其他限制条件，比如基于环境变量userenv。

3.3 小结

本章详细介绍了Oracle数据库为识别可重现的性能问题而提供的跟踪和探查功能。特别介绍了SQL跟踪及其相关工具，以及两个通过包dbms_hprof和dbms_profiler而具体化了的PL/SQL探查器。借助这些工具，当你尝试诊断由SQL语句或PL/SQL代码引起的性能下降时，就不会感到手忙脚乱。

当你无法重现问题，或者在问题发生时才不得不分析它时，本章介绍的方法在大多数时候是没用的。对于这些情形，你可以应用接下来在第4章中介绍的功能。

第4章

实时分析不可重现的问题

实时分析性能问题可以从动态性能视图中获取关键信息。在众多视图中，找到正确的视图并用合适的排序来查询，是有效确定性能问题的关键。为了能找到正确的动态性能视图，你需要考虑下面这个关键问题：

我能使用Diagnostics Pack和Tuning Pack选件吗？

这个问题根本就不是技术问题。然而回答这个问题是很必要的，因为只有在你有对应的许可时，才可以使用某些动态性能视图和数据库特性（实时分析的关键特性是活动会话历史和实时监控）。请注意，Diagnostics Pack选件是Tuning Pack选件的先决条件。同时所有的选件只在企业版中可用，标准版并不支持。有关许可的详细信息，请参考*Oracle Database Licensing Information*手册。

提示 如果没有Diagnostics Pack选件和Tuning Pack选件的许可，从11.1版本开始可以设置相应的初始化参数control_management_pack_access。企业版的默认值是diagnostic+tuning。这代表两个选件都被启用。其他可用的值有diagnostic和none。前者仅仅启用Diagnostics Pack选件，而后者禁用两个选件，并且这也是标准版的默认值。为这个初始化参数设置合适的值有两个好处。第一，禁止一些特性可以防止不必要的开销。第二，当使用Enterprise Manager时，这两个选件对应的页面会无法打开，这样就不会违反许可协议。

分析的步骤与能否使用这些可选特性无关。让我们用一张分析路线图来讨论一下。

4.1 分析路线图

图4-1显示了性能分析所需要的步骤。首先，你需要检查数据库服务器的负载情况，特别是执行两个检查。第一，确定数据库服务器是否是CPU bound。如果是，那么许多统计信息会被人为地扩大。因此，首要目标是要找出一种方法来减小CPU使用率。第二，检查消耗大量CPU的进程是否与数据库实例无关。如果是的话，那么无法找到引起性能问题的原因可能是你的关注点错了。

检查了数据库服务器负载后，你有三个选择。你需要问自己下面这个问题来决定选择哪个：

我的目标是什么？是单条SQL语句、单个会话还是整个系统？

你可以有针对性地处理正在经历性能问题的某条SQL语句或会话。实际上，正如第1章所述，你应该尽可能关注具体的问题。然而，当获取不到关键信息时，你需要继续在系统级别进行分析。例如，

当发生生产环境运行缓慢并且原因未知时，你的目标不再是处理某个业务的性能问题（当然，这始终是你努力的方向），而是想办法降低系统的负载。

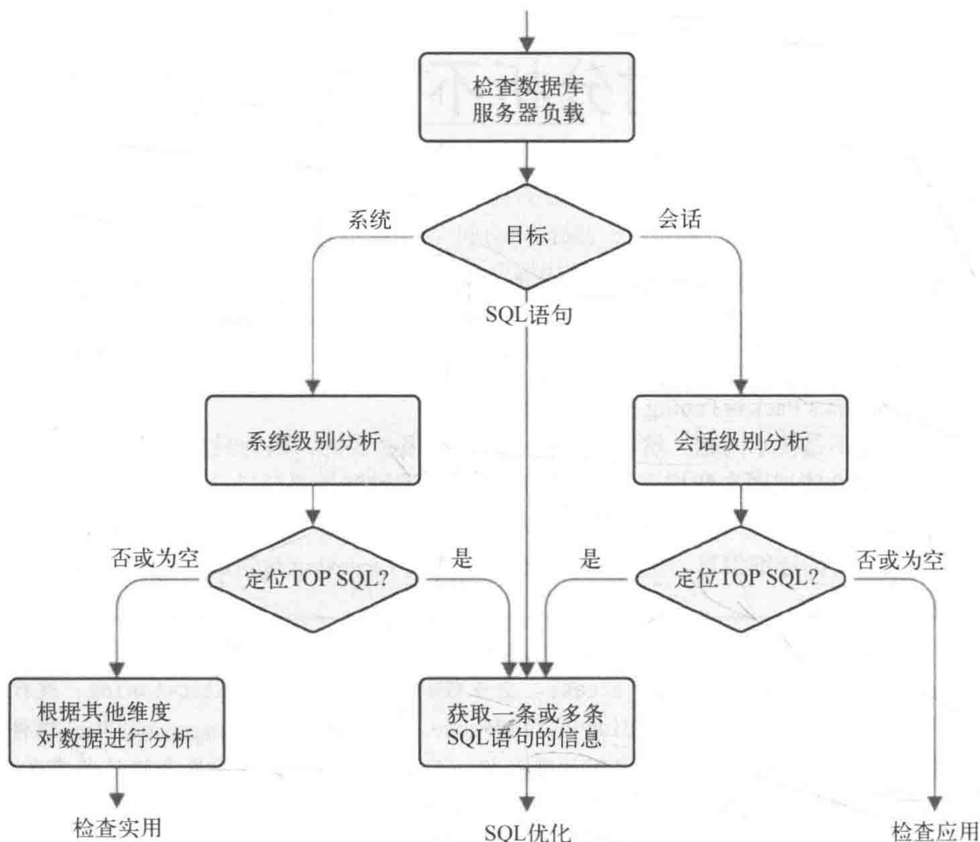


图4-1　实时分析不可重现问题的路线图

如果你继续在系统或会话层面分析，你的目标应该是找出是否有一小部分SQL语句（比如12个）造成了大量的负载。例如，你发现85%的负载是由7条SQL语句造成的，应该能很清楚地定位负载较多的SQL。然而，如果排名前10位的SQL语句只占用了25%的负载，那么它们就不值得你浪费时间去关注。

你同样应该检查是否有一小部分“组件”（比如，会话、模块或者客户端）占用了大量的负载。如果是，应该仅针对“组件”进行分析。总之，如果系统层面的分析并没有找出占用负载较多的SQL语句，你应该考虑检查应用是否在有效运行。如果应用代码无法进行分析（或者修改），或者检查后并没有令人满意的结果，那么你能考虑的就只剩下资源管理了。简单来说，你有两个选择：第一，明确应用的哪些部分（比如，一些会话或用户）使用了比其他部分更多的资源；第二，给应用更多的资源（比如硬件）。当然后者应该是你最后才会去考虑的。

思考这样一个特别的案例，你分析的系统或会话几乎是空闲的。这里的空闲，是指大部分处理时间都不是花费在数据库系统里。例如，你看到一个报表运行了13分钟，但数据库引擎只花了42秒来处

理相关SQL语句，当然这与处理了多少条SQL语句无关，这时关注数据库层是没用的。很明显，瓶颈并不在数据库层。这时也需要检查程序或其他支撑程序的部分。

针对你找到的每条占用负载较多的SQL语句，都需要收集执行计划、关键运行时统计（比如已处理的行数和CPU使用率的数量）以及已经历的等待事件，这与你关注的是单独的SQL语句、单个会话还是整个系统无关。下一节将介绍如何找出你需要知道的重要动态性能视图的基本信息。然后本章将详细解释如何使用动态性能视图提供的信息来实时分析性能问题。

4.2 动态性能视图

Oracle数据库利用动态性能视图来展现一些属于内存或数据库文件的数据结构内容。换句话说，即使它们看起来像普通的表，基础结构保存的数据却完全不同。这些结构以视图的形式展现出来，方便用户使用SQL语句获取数据。

数据库引擎时常会修改动态性能视图依赖的数据结构，因此动态性能视图提供的数据也是实时变化的。请注意并不是每个动态性能视图都以同样的方式更新。例如，其中一些视图持续更新，而其他的视图每5秒才更新一次。

警告 查询动态性能视图并不能保证一致读。因此，不要让小错误或不一致影响你。

我通常会引用v$前缀的视图来介绍动态性能视图。如果你使用的是RAC环境，请注意，v$视图只会显示你当前连接实例的信息。如果需要其他实例的信息，你需要使用带gv$前缀的全局视图。gv$视图的结构与v$视图相同。通常情况下，唯一不同的是gv$视图会多出一列（inst_id）用来标识数据库实例。

一些动态性能视图提供的统计信息依赖于初始化参数timed_statistics，这个参数可以设置成TRUE或FALSE。如果设置成TRUE，计时信息生效。如果设置成FALSE，则看不到这些统计信息。然而，根据你工作的平台不同，这些信息也可能存在。timed_statistics的默认值跟另一个初始化参数有关：statistics_level。如果将statistics_level设置成basic，那么timed_statistics的默认值为FLASE，否则其默认值为TRUE。由于这两个参数的默认值已经很适合，因此不建议在系统级别更改它们。

数据库里存在许多动态性能视图。下面介绍一些处理性能问题时经常会用到的视图。

4.2.1 操作系统统计信息

如果你无法访问数据库服务器，但又想得到一些操作系统级别的核心性能指标（如CPU和内存使用率），你可以查询v$osstat获取你想要的信息。并且多亏了comments列，你可以很清晰地知道每行值的含义。请注意有些列的值是从数据库实例启动时开始累积的（比如，所有提供计时信息的统计信息），其他则是常量（比如，socket数、CPU数和CPU核数）或者是当前值（比如，空闲内存的总量）。同时要注意，根据数据库版本和平台的不同，视图的实际内容也会不同。下面的例子是在Linux平台上针对12.1版本执行的查询：

```
SQL> SELECT stat_name, value, comments
  2  FROM v$osstat
```

```
STAT_NAME                         VALUE COMMENTS
------------------------ ------------ -------------------------------------------------------
NUM_CPUS                              8 Number of active CPUs
IDLE_TIME                      29648458 Time (centi-secs) that CPUs have been in the idle state
BUSY_TIME                       6348349 Time (centi-secs) that CPUs have been in the busy state
USER_TIME                       4942391 Time (centi-secs) spent in user code
SYS_TIME                        1336523 Time (centi-secs) spent in the kernel
IOWAIT_TIME                     3806135 Time (centi-secs) spent waiting for IO
NICE_TIME                         22373 Time (centi-secs) spend in low-priority user code
RSRC_MGR_CPU_WAIT_TIME            14195 Time (centi-secs) processes spent in the runnable state
                                        waiting
LOAD                                  1 Number of processes running or waiting on the run queue
NUM_CPU_CORES                         8 Number of CPU cores
NUM_CPU_SOCKETS                       2 Number of physical CPU sockets
PHYSICAL_MEMORY_BYTES       12619522048 Physical memory size in bytes
VM_IN_BYTES                           0 Bytes paged in due to virtual memory swapping
VM_OUT_BYTES                          0 Bytes paged out due to virtual memory swapping
FREE_MEMORY_BYTES            1529409536 Physical free memory in bytes
INACTIVE_MEMORY_BYTES        2112192512 Physical inactive memory in bytes
SWAP_FREE_BYTES              8603631616 Swap free in bytes
TCP_SEND_SIZE_MIN                  4096 TCP Send Buffer Min Size
TCP_SEND_SIZE_DEFAULT             16384 TCP Send Buffer Default Size
TCP_SEND_SIZE_MAX               4194304 TCP Send Buffer Max Size
TCP_RECEIVE_SIZE_MIN               4096 TCP Receive Buffer Min Size
TCP_RECEIVE_SIZE_DEFAULT          87380 TCP Receive Buffer Default Size
TCP_RECEIVE_SIZE_MAX            6291456 TCP Receive Buffer Max Size
GLOBAL_SEND_SIZE_MAX            1048576 Global send size max (net.core.wmem_max)
GLOBAL_RECEIVE_SIZE_MAX         4194304 Global receive size max (net.core.rmem_max)
```

这些信息对找出数据库服务器上是否存在消耗CPU资源的其他应用特别有帮助。为达到这个目的，你需要根据时间模块统计信息中的BUSY_TIME值来对比CPU使用率。如果这两个值相近，你就可以知道你连接的数据库实例占用了大多数CPU资源。

4.2.2 时间模型统计信息

通过查看时间模型统计信息，你可以知道数据库引擎代表应用进行哪种类型的处理。时间模型统计信息的目的是展示执行关键操作所花费的时间统计，比如打开新会话、解析SQL语句以及利用数据库引擎（SQL、PL/SQL、JAVA和OLAP）处理调用。另外也会提供一些关于后台处理的图表。

由两个独立的树形结构组织而成的一小部分图表，构成了时间模型统计信息：其中一个是数据库实例自身的后台处理，另外一个是前台处理（应用执行的处理）。图4-2和图4-3不仅显示了后台处理与前台处理的相关统计信息，也展示了它们之间的关系。比如，根据图4-3，parse time elapsed是DB time的子节点，也是hard parse elapsed time的上层节点。每项统计信息基本上可以根据名字知道其意义。具体的描述请参照*Oracle Database Reference*手册中的“V$SESS_TIME_MODLE”部分。

```
background elapsed time
└background cpu time
  └RMAN cpu time (backup/restore)
```

图4-2 后台处理时间模型统计信息的树形结构

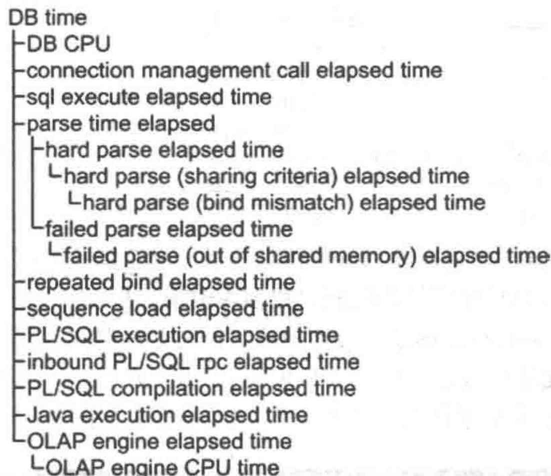


图4-3 前台处理时间模型统计信息的树形结构

由于时间模型统计信息被分成了两个树形结构，所以可以对DB time和background elapsed time进行求和来统计处理的合计时间。注意，这两个统计信息都包含CPU使用和除了空闲等待级别等待事件之外的其他所有等待事件的总和（下一节会介绍关于等待级别的详细信息）。

树形结构中子节点记录的时间会包含在父节点中。但这并不代表父节点会记录所有子节点记录的时间。实际上，有些操作不会只与单独的一个子节点相关联，甚至有些操作都不会归于任何子节点。

时间模型统计信息分别由v$sys_time_model和v$sess_time_model视图来提供系统级别和所有连接会话的信息。此外，在12.1多租户环境下，v$con_sys_time_model视图会显示容器级别的统计信息。在这些动态性能视图中，有下面两个关键列。

- stat_name标识统计信息。
- value提供对应组件（数据库实例、进程或容器）从初始化开始累积的总时间（单位：微秒）。

对于会话级别的统计信息（v$sess_time_model），也有一列（sid）用来标识相关会话。并且在12.1多租户环境下，也同样存在一列（con_id）用来标识容器。

以下查询基于v$sess_time_model视图，显示某进程从启动开始花费的处理时间（97.3%的时间用来执行SQL语句）：

```
SQL> WITH
  2    db_time AS (SELECT sid, value
  3                FROM v$sess_time_model
  4                WHERE sid = 42
  5                AND stat_name = 'DB time')
  6  SELECT ses.stat_name AS statistic,
  7         round(ses.value / 1E6, 3) AS seconds,
  8         round(ses.value / nullif(tot.value, 0) * 1E2, 1) AS "%"
  9  FROM v$sess_time_model ses, db_time tot
 10  WHERE ses.sid = tot.sid
 11  AND ses.stat_name <> 'DB time'
 12  AND ses.value > 0
 13  ORDER BY ses.value DESC;
```

```
STATISTIC                                    SECONDS     %
-------------------------------------------- ------- -----
sql execute elapsed time                      99.437  97.3
DB CPU                                          4.46   4.4
parse time elapsed                             0.308   0.3
connection management call elapsed time        0.004   0.0
PL/SQL execution elapsed time                  0.000   0.0
repeated bind elapsed time                     0.000   0.0
```

请注意，在本例中，百分比是根据DB time的值计算的，这个值是数据库引擎处理用户调用花费的所有时间。由于DB time只计算数据库处理时间，数据库引擎等待用户调用的时间花费并不包括在内。因此，仅根据时间模型统计信息提供的信息并不能确定问题是在数据库里还是数据库之外。此外，仅根据时间模型统计信息也无法解释已用时间和CPU时间的区别（比如，上例中只有4.4%的时间用在CPU上）。要想确切知道到底发生了什么，就需要等待级别和等待事件（见下一节）的信息。

平均活动会话数

平均活动会话数（Average Number of Active Session，AAS）是系统级别DB time的增长率。比如，如果数据库实例的BD time在60秒内增长了1860秒，那么平均活动会话数为31（1860/60）。这表示在60秒内，大约有31个进程在处理用户调用。

系统级别的平均活动会话数是一个重要指标，因为它能告诉你系统的负载情况。一般来说，当系统几乎空闲时，这个值要比CPU核数低得多。相反，若该值比CPU核数高许多，则表示系统非常繁忙。不得不说由于这是个平均值，会有很多信息看不到。因此当只根据该信息来诊断时请格外注意，尤其当持续时间不止几分钟时。

DB time的增长率也可以用来计算单独的会话。那样的话，它只能是介于0和1之间的某个值，用来表示会话活动的程度。如果值是0，代表进程彻底空闲。换句话说，数据库引擎没有做处理。如果值为1，代表进程正忙于处理用户调用。

4.2.3 等待级别和等待事件

基于时间模型统计信息，不仅可以确定数据库实例（会话或容器）花费了多少时间处理，也能知道这个处理使用了多少CPU。当这两个值相等时，代表数据库实例并没有经历类似磁盘I/O操作、网络回路或者锁的等待。然而，当这两个值不同时，为了分析性能问题，你需要知道服务器进程正在等待哪些资源。而这个信息就来自等待事件。

由于等待事件非常多（在12.1版本中有超过1500个），为了简化分析产品的资源使用，等待事件被分成13个等待级别（注意，10.2版本中是12个）。通过查询v$event_name视图可以得知当前版本有哪些等待事件及其等待级别。例如，以下查询显示了在12.1.0.1版本中存在的等待级别，每个级别包含的等待事件数以及一些等待事件（Commit）所属的等级级别：

```
SQL> SELECT wait_class, count(*)
  2  FROM v$event_name
  3  GROUP BY rollup(wait_class)
  4  ORDER BY wait_class;
```

```
WAIT_CLASS      COUNT(*)
-------------- --------
Administrative       57
Application          17
Cluster              57
Commit                4
Concurrency          34
Configuration        26
Idle                119
Network              28
Other              1123
Queueing              9
Scheduler            10
System I/O           34
User I/O             51
                   1569

SQL> SELECT name
  2  FROM v$event_name
  3  WHERE wait_class = 'Commit';

NAME
----------------------------------
remote log force - commit
log file sync
nologging standby txn commit
enq: BB - 2PC across RAC instances
```

v$system_wait_class和v$session_wait_class视图分别记录了系统级别与所有连接会话的等待事件级别。此外，在12.1多租户环境下，v$con_system_wait_class显示容器级别的统计信息。这些动态性能视图有下面三个关键列。

- wait_class标识等待级别。
- toal_waits提供对应组件（数据库实例、会话或容器）从初始化开始累积的等待事件数。
- time_waited提供对应组件（数据库实例、会话或容器）从初始化开始累积的总等待时间（单位：百分之一秒）。

当然，对于会话级别的统计信息（v$session_wait_class），同样存在列（sid）用来标识对应的会话，并且在12.1多租户环境下，存在列（con_id）用来标识容器。接下来的例子使用与之前例子一样的会话（在CPU上花费4.4%的DB time）。用这个例子说明，如何对会话执行处理以生成一个简要的资源使用分析。可以在系统级别和容器级别使用类似的查询。只需要将查询涉及的动态性能视图更改为对应的级别即可。

```
SQL> SELECT wait_class,
  2         round(time_waited, 3) AS time_waited,
  3         round(1E2 * ratio_to_report(time_waited) OVER (), 1) AS "%"
  4  FROM (
  5    SELECT sid, wait_class, time_waited / 1E2 AS time_waited
  6    FROM v$session_wait_class
  7    WHERE total_waits > 0
  8    UNION ALL
```

```
  9    SELECT sid, 'CPU', value / 1E6
 10    FROM v$sess_time_model
 11    WHERE stat_name = 'DB CPU'
 12  )
 13  WHERE sid = 42
 14  ORDER BY 2 DESC;

WAIT_CLASS    TIME_WAITED     %
------------- ----------- -----
Idle               154.77  60.2
User I/O            96.99  37.7
CPU                  4.46   1.7
Commit               0.85   0.3
Network              0.04   0.0
Configuration        0.03   0.0
Concurrency          0.02   0.0
Application          0.01   0.0
```

即便已经开始基于等级级别的资源使用分析，大多数时候你仍然需要准确的信息。你需要等待事件。为此，数据库引擎分别通过v$system_event和v$session_event视图来提供系统级别和所有连接会话的等待事件信息。此外，在12.1多租户环境下，v$con_system_event视图提供容器级别信息。以下查询用来说明如何对会话执行处理以生成详细的资源使用分析（可以在系统级别和容器级别使用类似的查询。只需要将查询涉及的动态性能视图更改为对应的级别），使用的是与前面例子相同的会话。

```
SQL> SELECT event,
  2         round(time_waited, 3) AS time_waited,
  3         round(1E2 * ratio_to_report(time_waited) OVER (), 1) AS "%"
  4  FROM (
  5    SELECT sid, event, time_waited_micro / 1E6 AS time_waited
  6    FROM v$session_event
  7    WHERE total_waits > 0
  8    UNION ALL
  9    SELECT sid, 'CPU', value / 1E6
 10    FROM v$sess_time_model
 11    WHERE stat_name = 'DB CPU'
 12  )
 13  WHERE sid = 42
 14  ORDER BY 2 DESC;

EVENT                          TIME_WAITED     %
------------------------------ ----------- -----
SQL*Net message from client        154.790  60.2
db file sequential read             96.125  37.4
CPU                                  4.461   1.7
log file sync                        0.850   0.3
read by other session                0.734   0.3
db file parallel read                0.135   0.1
SQL*Net message to client            0.044   0.0
cursor: pin S                        0.022   0.0
enq: TX - row lock contention        0.011   0.0
Disk file operations I/O             0.001   0.0
latch: In memory undo latch          0.001   0.0
```

在上面的输出中，你可以注意到DB time只占用了总执行时间的39.8%（100–60.2）。实际上，剩下的60.2%被空闲等待事件占用（SQL*Net message from client）。这表示在60.2%的时间里，数据库引擎在等待应用提交作业。这个资源使用分析提供的另一个重要的信息是，当数据库引擎处理用户调用时，它总是单块读来执行磁盘I/O操作（db file sequential read）。所有其他的等待事件和CPU使用率都可忽略不计。

对于一些等待事件，比如与磁盘I/O操作有关的，你或许想要知道平均延迟的信息。实际上，如果你有这些信息，就可以对比当前性能与预期性能（你应该知道用来存储数据库的磁盘I/O子系统的预期性能）。比如，可以基于视图（比如v$system_event）执行查询来计算某一等待事件的平均延迟。


```
SQL> SELECT time_waited_micro/total_waits/1E3 AS avg_wait_ms
  2  FROM v$system_event
  3  WHERE event = 'db file sequential read';

AVG_WAIT_MS
-----------
 9.52927176
```

上面这个查询计算出来的平均值隐藏了很多信息，Oracle数据库提供了一个视图，该视图可以在系统级别为每个等待事件提供直方图。这个视图就是v$event_histogram。它有下面三个关键列。

- event是等待事件的名称。
- wait_time_milli代表每个直方图桶的上限值（不包含在内）。
- wait_count是与等待事件关联的直方图桶数。

比如，接下来的查询显示多数（45.7%）等待事件在4毫秒和8毫秒的桶内，约24%（3.27+2.75+18.37）在小于4毫秒的桶内，约10%（5.96+2.66+1.34+0.17+0.01）在16毫秒以及更大的桶内。

```
SQL> SELECT wait_time_milli, wait_count, 100*ratio_to_report(wait_count) OVER () AS "%"
  2  FROM v$event_histogram
  3  WHERE event = 'db file sequential read';

WAIT_TIME_MILLI WAIT_COUNT      %
--------------- ---------- ------
              1     348528   3.27
              2     293508   2.75
              4    1958584  18.37
              8    4871214  45.70
             16    2106649  19.76
             32     635484   5.96
             64     284040   2.66
            128     143030   1.34
            256      18041   0.17
            512        588   0.01
           1024        105   0.00
           2048          1   0.00
```

上面的信息主要显示最大值是多少。在本例中，一些磁盘I/O操作远远超出预期（在64毫秒的桶与2秒的桶之间）。尽管这样的操作不多，却能表明要么磁盘I/O系统（或其中一个组件）太小，要么配置或硬件有问题。

4.2.4　系统和会话统计信息

除了时间模块统计信息和等待事件，数据库引擎同样会记录数百个（在12.1版本中有超过850个）附加统计信息，比如某一操作执行的次数或某一函数处理的数据量。可以分别在v$sysstat和v$sesstat视图中查到系统级别和所有连接会话的相关信息。此外，在12.1多租户环境中，也可以在v$con_sysstat视图中找到容器级别的信息。

在v$sysstat视图中有下面两个关键列。

- ❑ name标识统计信息（大部分的简要描述请参考*Oracle Database Reference*手册）。
- ❑ value提供与统计信息相关的指标。在大多数情况下，value显示的是从数据库实例启动开始的累计值，但并不是所有的统计信息都是这样。

让我们来看两个查询，这两个例子会显示动态性能视图，比如v$sysstat提供的信息。第一个查询基于持续增加的计数器返回统计信息。在这里，这些计数器代表logon数、commit数和数据库实例启动后在内存中的排序数。

```
SQL> SELECT name, value
  2  FROM v$sysstat
  3  WHERE name IN ('logons cumulative', 'user commits', 'sorts (memory)');

NAME                 VALUE
----------------- --------
logons cumulative     1422
user commits       1298103
sorts (memory)      770169
```

第二个查询返回的统计信息显示磁盘I/O操作处理的总数据量。

```
SQL> SELECT name, value
  2  FROM v$sysstat
  3  WHERE name LIKE 'physical % total bytes';

NAME                              VALUE
------------------------- -----------
physical read total bytes   9.1924E+10
physical write total bytes  4.2358E+10
```

v$con_sysstat视图与v$sysstat视图的结构一样。然而，v$sesstat视图有很大不同。尽管有列（sid）用来标识统计信息所属的会话，但没有提供name列。为了获得统计信息的名称，就必须使用另一个包含所有统计信息列表的视图v$statname与v$sesstat进行联合查询。以下查询展示了如何利用这两个视图来获取当前会话的PGA内存使用率信息（会返回两个值：当前内存使用的总数和会话初始化后分配的最大内存数）：

```
SQL> SELECT sn.name, ss.value
  2  FROM v$statname sn, v$sesstat ss
  3  WHERE sn.statistic# = ss.statistic#
  4  AND sn.name LIKE 'session pga memory%'
  5  AND ss.sid = sys_context('userenv','sid');

NAME                         VALUE
--------------------- --------
```

```
session pga memory       1723880
session pga memory max   2313704
```

为了避免基于sid列的限制，以上查询可以使用v$mystat视图。实际上，针对会话查询，还可以使用提供相同信息的v$sesstat。唯一不同的是这个视图只显示当前会话的统计信息。

提示 大多数情况下，要开始分析，首先需要把响应时间分解成CPU消耗与等待事件。如果一个会话总是占用CPU，没有任何等待事件，会话统计信息会有助于了解当前会话到底在做什么。

4.2.5 度量值

前几节介绍的动态性能视图里提供的多数统计信息值都是累积的。以此为基础，数据库引擎计算出一个度量值（根据版本的不同，在200~300之间），而这个值对监控特别有用。这些值在v$metricname视图中列出。比如，从11.2版本之后，会有一个值用来显示数据库服务器每秒的CPU使用（基于OS统计信息）。以下查询展示了这个值在v$metricname视图中的内容：

```
SQL> SELECT metric_id, metric_unit, group_id, group_name
  2  FROM v$metricname
  3  WHERE metric_name = 'Host CPU Usage Per Sec';

METRIC_ID METRIC_UNIT             GROUP_ID GROUP_NAME
--------- ----------------------- -------- ------------------------------
     2155 CentiSeconds Per Second        2 System Metrics Long Duration
     2155 CentiSeconds Per Second        3 System Metrics Short Duration
```

正如你在上面这个查询输出中所看到的，一个度量值有一个ID、一个衡量单位和它所在组的ID和名称（本例它属于两个组）。

请注意度量值是基于若干衡量单位计算出来的。其中一些，像上面例子中的，代表使用率或每秒的事件数。其他的则是根据每个事务、请求、调用或绝对值的平均值计算出来的。

度量值关联的组定义了计算间隔和信息提供的时长。如果一个度量值关联两个组，正如上面的例子那样，这表示数据库引擎分别计算两个度量值，并且它们都有各自的间隔和保存期。v$metricgroup视图提供了关于组的信息。下面这些组存在于12.1版本中（其他版本中组的数量可能不同）：

```
SQL> SELECT *
  2  FROM v$metricgroup
  3  ORDER BY group_id;

  GROUP_ID NAME                             INTERVAL_SIZE MAX_INTERVAL
---------- -------------------------------- ------------- ------------
         0 Event Metrics                             6000            1
         1 Event Class Metrics                       6000           60
         2 System Metrics Long Duration              6000           60
         3 System Metrics Short Duration             1500           12
         4 Session Metrics Long Duration             6000           60
         5 Session Metrics Short Duration            1500            1
         6 Service Metrics                           6000           60
         7 File Metrics Long Duration               60000            6
```

```
  9 Tablespace Metrics Long Duration        6000          0
 10 Service Metrics (Short)                  500         24
 11 I/O Stats by Function Metrics           6000         60
 12 Resource Manager Stats                  6000         60
 13 WCR metrics                             6000         60
 14 WLM PC Metrics                           500         24
```

interval_size列显示度量值所关联的组的计算间隔，以百分之一秒为单位。例如，System Metrics Long Duration组的度量值每60秒计算一次。max_interval列显示该组间隔保留的最大值。例如，System Metrics Long Duration组的最大值为60。因此对于这个组，前一个小时的信息都是可用的。

度量值自身的值来自若干视图。实际上，有些视图是特别针对一些度量值组的。例如，对于组2和组3，v$sysmetric和v$sysmetric_history视图分别显示了当前值和历史值。简单来说，对于大多数度量值，都可以使用v$metric和v$metric_history视图。举例说明，下面的查询显示Host CPU Usage Per Sec的当前度量值（注意，第一个度量值计算花费了60秒，第二度量值只花费了15秒）：

```
SQL> SELECT begin_time, end_time, value, metric_unit
  2  FROM v$metric
  3  WHERE metric_name = 'Host CPU Usage Per Sec';

BEGIN_TIME          END_TIME                 VALUE METRIC_UNIT
------------------- ------------------- ---------- ----------------------
2014-04-28 01:56:00 2014-04-28 01:57:00 168.137173 CentiSeconds Per Second
2014-04-28 01:56:45 2014-04-28 01:57:00 159.786951 CentiSeconds Per Second
```

4.2.6　当前会话状态

通过v$session视图，不仅可以知道有哪些会话存在，还可以知道它们现在都在做什么。由于这个动态性能视图包含了太多列（例如，10.2.0.5版本中为82列，12.1.0.1版本中为101列），这里不会一一介绍（更多信息请参考*Oracle Database Reference*手册）。下面列出可以从v$session视图中获取的最重要的信息以及这些信息所在的列。

- 会话的标识（sid、serial#、saddr和audsid），会话是属于BACKGROUND会话还是USER会话（type），以及会话进行初始化的时间（logon_time）。
- 打开会话的用户的标识（username和user#）、当前模式（schemame）和用于连接到数据库引擎的服务的名称（service_name）。
- 使用会话的应用（program）、启动会话所在的机器（machine）、会话的进程ID（process）以及启动会话的操作系统用户的名称（osuser）。
- 服务器端进程的类型（server）（可以是DEDICATED、SHARED、PSEUDO、POOLED或NONE）以及服务器端进程的地址（paddr）。
- 当前活动事务的地址（taddr）。
- 会话状态（status）（可以是ACTIVE、INACTIVE、KILLED、SNIPED或CACHED）以及这个状态持续了多少秒（last_call_et）。处理性能问题时，通常只关注ACTIVE的会话。
- 正在执行的SQL语句的类型（command）、与SQL语句相关的游标的标识（sql_address、sql_hash_value、sql_id和sql_child_number）、执行的开始时间（sql_exec_start）以及SQL

语句的执行ID（sql_exec_id）。执行ID是一个整数值，与sql_exec_start一起标识出某个特定执行。由于同样的游标每秒会被执行多次，这会变得很重要（注意sql_exec_start列的数据类型是DATE）。

- 执行过的前一个游标的标识（prev_sql_address、prev_hash_value、prev_sql_id和prev_child_number）、前一个执行的开始时间（prev_exc_start）以及前一个游标的执行ID（prev_exec_id）。
- 如果执行的是PL/SQL调用，那么该信息包括，被调用的顶层程序与子程序的标识（plsql_entry_object_id和plsql_entry_subprogram_id），以及当前正在执行的顶层程序和子程序（plsq_object_id和plsql_subprogram_id）。注意，如果会话正在执行某个SQL语句，则会将plsql_object_id和plsql_subprogram_id设置为NULL。
- 会话属性（client_identifier、module、action和client_info）（如果使用会话的应用设置这些属性）。
- 如果会话当前正在等待（这种情况下会将state列设置为WAITING），那么该信息包括，会话正在等待的等待事件的名称（event）、其等待级别（wait_class和wait_class#）、关于等待事件的详细信息（p1text、p1、p1raw、p2text、p2、p2raw、p3text、p3和p3raw），以及会话已经等待该等待事件的时间（seconds_in_wait，自11.1版本起为wait_time_micro）。注意如果state列不是WAITING，那么表示会话在使用CPU（如果status列等于ACTIVE）。这种情况下，与等待事件相关的列会包含关于上一次等待的信息。
- 会话是否被另一个会话所阻止（如果是，则会将blocking_session_status设置为VALID）；如果会话正在等待，那么是哪个会话正在阻止它（blocking_instance和blocking_session）。
- 如果会话当前被阻止并且正在等待某个特定行（例如，等待某个行锁），那么该信息是会话当前正在等待的行的标识（row_wait_obj#、row_wait_file#、row_wait_block#和row_wait_row#）。如果会话未在等待某个被锁定的行，那么row_wait_obj#列等于值-1。

除了v$session视图之外，还有专门提供特定信息的其他动态性能视图。比如，v$session_wait视图仅提供与等待事件相关的列，而v$session_blockers视图仅提供与被阻止会话相关的列。

4.2.7 活动会话历史

上一节介绍过，通过v$session视图可以知道所有连接会话的当前状态。即使这样的信息有用，却也并不足以用来分析性能问题。实际上，想要分析成功，就必须知道一个会话在一段时间内做了什么，而不是仅仅在某一时刻做了什么。这就是*活动会话历史*（ASH）的作用，它可以帮助你获取会话状态的历史信息。

注意 Diagnostics Pack选件必须有许可才能使用ASH。如果control_management_pack_access默认设置为none，ASH会被禁用。

与SQL跟踪相比，ASH的主要优势在于ASH总是处于启用状态，因此可以在需要时随时查看。正是由于这个原因，它对于不能重现的性能问题分析非常有用。你仅需要等到系统经历性能问题后去分析ASH的信息即可。

为了生成ASH的历史信息，后台进程（MMNL）会在每秒执行以下三个操作。

- 对所有会话状态取样（它的执行内容类似查询v$session视图）。
- 忽略等待空闲等待级别事件的会话数据。
- 把余下的数据存入SGA的内存缓冲区中。

图4-4为示例图解。你可以注意到会话1在等待用户I/O级别的事件，而会话2在使用CPU，会话3和会话4处于空闲状态。

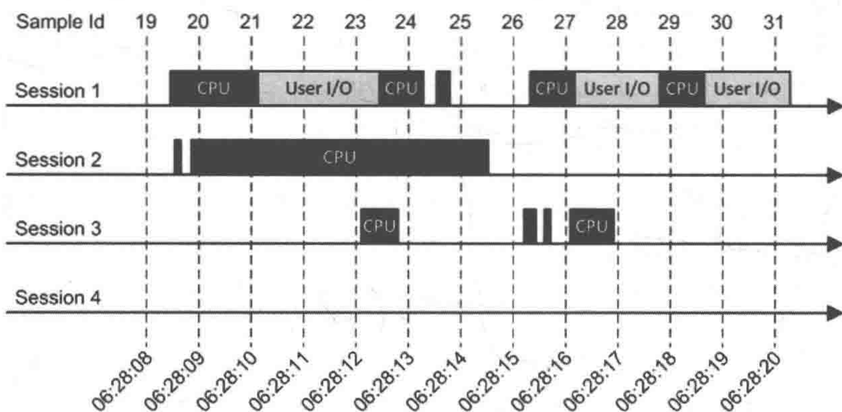


图4-4　MMNL进程取样，一秒内所有会话的状态

如图4-4所示，MMNL取样的进程产生的数据与表4-1的汇总数据类似。这里需要注意两点。首先，活动会话历史里总是会记录至少持续一秒的操作。其次，即使会话3有过一些活动，但也不会记录在活动会话历史中。

表4-1　针对图4-4示例的负载，由MMNL生成的样例

样例ID	时间戳	会　话	SQL ID	活　动
20	06:28:09	1	gd90ygn1j4026	CPU
20	06:28:09	2	5m6mu5pd9w028	CPU
21	06:28:10	1	gd90ygn1j4026	CPU
21	06:28:10	2	5m6mu5pd9w028	CPU
22	06:28:11	1	gd90ygn1j4026	User I/O
22	06:28:11	2	5m6mu5pd9w028	CPU
23	06:28:12	1	gd90ygn1j4026	User I/O
23	06:28:12	2	5m6mu5pd9w028	CPU
24	06:28:13	1	7ztv2z24kw0s0	CPU
24	06:28:13	2	5m6mu5pd9w028	CPU
25	06:28:14	2	5m6mu5pd9w028	CPU
27	06:28:16	1	d9gdx5a4gc13y	CPU
28	06:28:17	1	1uaz41wrxw03k	User I/O
29	06:28:18	1	1uaz41wrxw03k	CPU
30	06:28:19	1	1uaz41wrxw03k	User I/O
31	06:28:20	1	1uaz41wrxw03k	User I/O

基于表4-1中的数据，可以派生出以下信息。

- 会话1活动时间占总时间的83%（12秒内10个取样）并且至少执行了4个不同的SQL语句。CPU占用了一半的处理时间（10秒内5个取样），另一半时间用来执行磁盘I/O操作。
- 会话2活动时间占总时间的50%（12秒内6个取样）。这期间CPU一直在执行SQL ID为5m6mu5pd9w028的SQL。
- 会话3和会话4处于100%空闲状态（12秒内0个取样）。

活动会话历史缓冲区

当数据库实例启动时，会在SGA中生成一个缓冲区用来存放活动会话历史。Oracle的设计目的是在内存中保存一个小时的活动。可以执行以下查询来获取缓冲区大小以及存放了多久的信息：

```
SQL> SELECT pool, bytes
  2  FROM v$sgastat
  3  WHERE name = 'ASH buffers';

POOL             BYTES
----------- ---------
shared pool  14680064

SQL> SELECT max(sample_time) - min(sample_time) AS interval
  2  FROM v$active_session_history;

INTERVAL
-----------------------
+000000000 02:25:30.293
```

可以通过v$active_session_history视图来查询存储在活动会话历史中的数据。视图中的许多列与v$session视图的意义相同（详细信息请参考4.2.6节）。就像v$session视图一样，v$active_session_history视图的列与使用版本有很大关系（比如，10.2.0.5版本有50列，12.1.0.1版本有101列）。与v$session视图相比，v$active_session_history视图少了一些列，其中一些列的内容也有所不同，还有一些列仅存在于v$active_session_history视图中。下面列出了仅存在于v$active_ session_history视图中的重要列或内容不同的列（更多信息请参考*Oracle Database Reference*手册）。

- sample_id是MMNL取样时的标识符。注意，在活动会话历史中，它并不用来标识一列。
- sample_time是MMNL取样时的时间戳。因此，可以基于此列重现每个会话的活动。
- session_state是会话的状态。该列值是WAITING或ON CPU。
- 如果会话在等待，那么time_waited列就是会话等待时间的总微秒数。为了防止一个等待事件会跨越两个或者更多的取样，实际等待时间会取自最后一个取样，对于其他的取样，time_waited列将为0。以下例子显示会话等待锁7.4秒：

```
SQL> SELECT sample_time, event, time_waited
  2  FROM v$active_session_history
  3  WHERE session_id = 137
  4  ORDER BY sample_time;
```

```
SAMPLE_TIME                EVENT                 TIME_WAITED
-------------------------- --------------------- -----------
...
27-APR-14 03.10.50.245 PM  enq: TM - contention            0
27-APR-14 03.10.51.245 PM  enq: TM - contention            0
27-APR-14 03.10.52.245 PM  enq: TM - contention            0
27-APR-14 03.10.53.245 PM  enq: TM - contention            0
27-APR-14 03.10.54.245 PM  enq: TM - contention            0
27-APR-14 03.10.55.245 PM  enq: TM - contention            0
27-APR-14 03.10.56.245 PM  enq: TM - contention            0
27-APR-14 03.10.57.245 PM  enq: TM - contention      7390676
...
```

- 对于执行的SQL语句，若干列提供执行计划信息。特别是它的值（sql_plan_hash_value）和活动操作（sql_plan_line_id、sql_plan_operation和sql_plan_options）。
- 从11.1版本之后，根据时间模型分析定义的分类，若干标记用来指出执行的操作（in_connection_mgmt、inparse、in_hard_parse、in_sql_execution、in_plsql_execution、in_plsql_rpc、in_plsql_comilation、in_java_execution、in_bind、in_cursor_close和11.2版本之后的in_sequence_load）。
- 对于并行执行的SQL语句，也有对应并行查询的信息（qc_instance_id、qc_session_id和11.1版本之后的qc_session_serial#）。

有了v$active_session_history视图中的信息，就可以分析数据库引擎在处理期间花费的时间。强调一下，之所以是统计分析是因为数据是基于取样的。因此，更多的取样才会产生更准确的结果。总之，由于每秒只会取样一次并且只保存一个活动会话，因此准确性比不上不基于取样的方法（比如SQL跟踪）。然而在很多时候，分析能提供足够的信息来指出性能问题的成因。

针对v$active_session_history视图的典型查询包含以下部分。

- 对sample_time进行限制来关注某个特定时间段。
- 根据一列或多列的聚合查询来获取处理的相关信息，如会话ID（session_id），执行游标的SQL语句（sql_id），或者执行处理的应用（program）。
- 取样的计数。由于每个取样都是一秒，取样的数量会接近DB time。

警告　要对DB time、CPU时间或等待事件的总时间估值，需对取样计数。但请注意，使用sum(time_waited)这样的简单表达式聚合数是错的。因为事件被取样的可能性与事件长度有关。

例如，以下查询返回的是五分钟内根据它们的DB time排序的排名前十的SQL语句（注意，比如排名第一的SQL语句执行了1008秒，占用了29.9%的DB time）。

```
SQL> SELECT activity_pct,
  2         db_time,
  3         sql_id
  4  FROM (
  5    SELECT round(100 * ratio_to_report(count(*)) OVER (), 1) AS activity_pct,
  6           count(*) AS db_time,
  7           sql_id
  8    FROM v$active_session_history
  9    WHERE sample_time BETWEEN to_timestamp('2014-02-12 22:12:30', 'YYYY-MM-DD HH24:MI:SS')
```

```
10                           AND to_timestamp('2014-02-12 22:17:30', 'YYYY-MM-DD HH24:MI:SS')
11     AND sql_id IS NOT NULL
12     GROUP BY sql_id
13     ORDER BY count(*) DESC
14   )
15   WHERE rownum <= 10;

ACTIVITY_PCT    DB_TIME SQL_ID
------------ ---------- -------------
        29.9       1008 c13sma6rkr27c
        11.3        382 0yas01u2p9ch4
        11.2        376 0y1prvxqc2ra9
         9.5        321 7hk2m2702ua0g
         8.2        277 bymb3ujkr3ubk
         7.8        263 8dq0v1mjngj7t
         5.8        196 8z3542ffmp562
         4.2        142 0bzhqhhj9mpaa
         2.8         93 5mddt5kt45rg3
         1.3         44 0w2qpuc6u2zsp
```

如果你使用Enterprise Manager 12*c*（Cloud Control或Express），则可以访问ASH Analytics获取活动会话历史的信息。使用ASH Analytics可以直接访问v$active_session_history视图执行分析而不用写SQL查询。可以简单地在数据库实例经历的时间线和负载概况的图表上选取一个时间段（如图4-5所示），选出想要聚合的一块或多块数据（下拉列表允许包含多达24个选择），然后选择数据显示的格式。

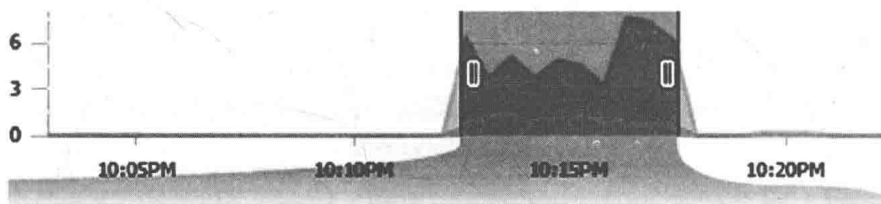


图4-5 在ASH Analytics上选择分析的时间段

注意 尽管ASH Analytics是Enterprise Manager的特性，但也需要在数据库端安装一些对象。这些对象只在11.2.0.4版本之后才会被默认安装。在之前的版本中是需要手工安装的。若未安装这些对象，使用ASH Analytics时，Enterprise Manager会建议你安装。注意，10.2版本不支持ASH Analytics。

可以根据以下三种主要的格式来排列数据。

- activity chart显示所选时间段中平均活动会话数的变化。图4-6显示了在图4-5中选择的5分钟内排名前十的SQL语句的活动会话数。

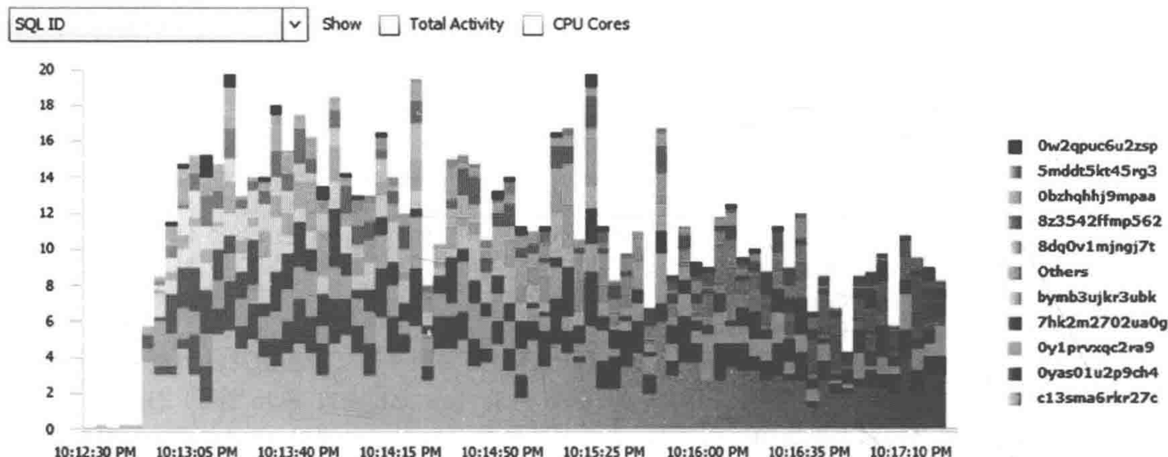


图4-6　activity chart显示图4-5选择的时间段中排名前十的SQL语句

- top consumer table显示选择的时间段中消耗最大的平均活动会话数。注意在这里可以选择与activity chart不同的区域。例如，图4-7显示了，在图4-5选择的时间段里，哪些SQL语句占用了最多的DB time。注意，将鼠标停在activity的条上时，也可以看到SQL语句执行的一些活动信息。例如，在图4-7中，排名第一的SQL语句花费了29%的时间用来等待用户I/O等待级别的事件。

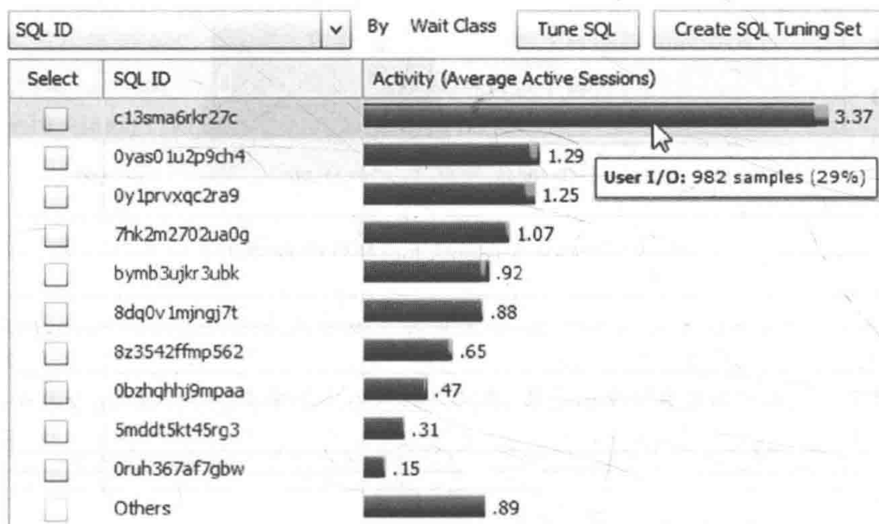


图4-7　top consumer表显示在图4-5选择的时间段里排名前十的SQL语句

- load map显示的信息与top consumer表很像，不同的是用了treemap而非表。例如，图4-8显示了与图4-7同时间段的数据。同样，也可以把鼠标悬停在load map的一个矩形上获取详细信息。

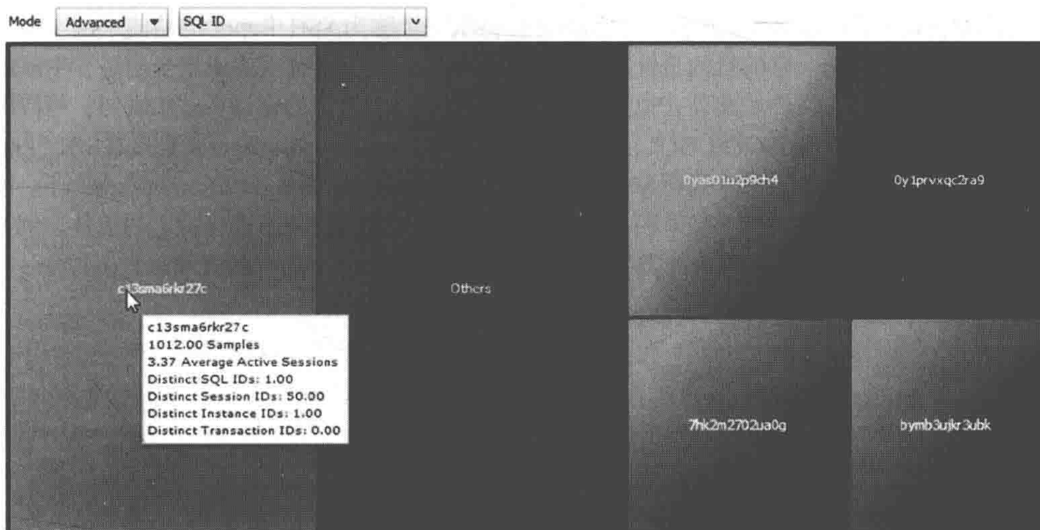


图4-8 load map表显示在图4-5选择的时间段里排名前五的SQL语句

尽管activity chart和load map可用来显示数据，但在ASH Analytics里它们扮演着其他重要的角色。实际上，可以通过选择一个或者多个top consumer来限制分析的数据。换句话说，可以利用它们定义过滤器来应用到图表里。例如，可以执行以下操作：

- 显示top SQL语句并选择最耗时的部分（c13sma6rkr27c）；
- 显示top wait class并选择最耗时的部分（用户I/O）；
- 显示top wait event并选择最耗时的部分（`db file sequential read`）；
- 显示top module。

图4-9显示了这些操作的结果。注意，load map顶端定义了过滤器。基于这个图，可以推断出选择时间段内的全部负载是由单个模块产生的（New Order）。

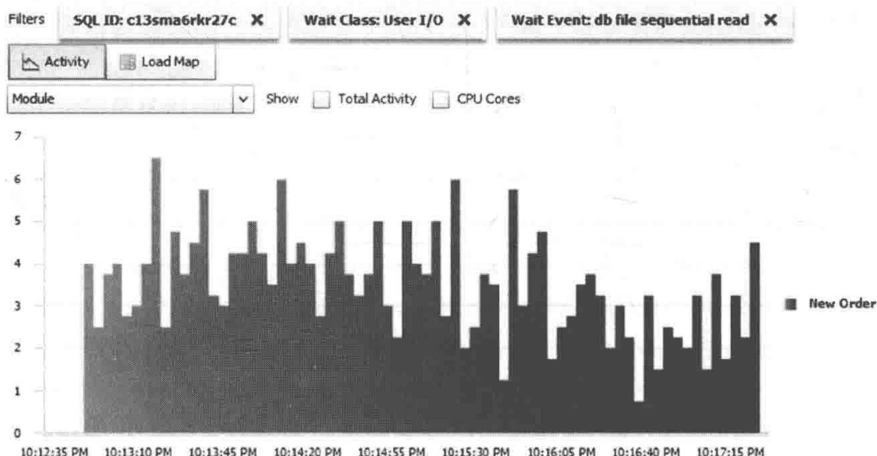


图4-9 activity chart显示了执行特定SQL语句时经历了特定等待事件的模块

无论你能否访问Enterprise Manager，Oracle都提供了一个称为ASH Report的功能来从ASH中抽取数据而不需要写SQL语句。它的目的是针对选择的时间段，根据若干维度生成聚合信息，生成的文件可以是文本文件或HTML文件。此外，也可以选择性地限制分析特定的会话、SQL语句、等待级别、服务、模块、动作、客户端ID或PL/SQL访问点。可以使用Enterprise Manager或直接在SQL*Plus里执行脚本ashrpt.sql或ashrpti.sql来运行ASH Report，脚本存放在$ORACLE_HOME/rdbms/admin下。这两个脚本的区别在于前者输入较少的参数。特别是在执行脚本时，你无法限制选择特定的组件。例如，执行ashrpt.sql脚本只会询问生成报告的类型（文本或HTML）、要分析的时间段和报告名。

```
SQL> @?/rdbms/admin/ashrpt.sql

Enter 'html' for an HTML report, or 'text' for plain text
Enter value for report_type: text

Enter value for begin_time: 02/12/14 22:12:30

Enter duration in minutes starting from begin time:
Enter value for duration: 5

The default report file name is ashrpt_1_0212_2217.txt. To use this name,
press <return> to continue, otherwise enter an alternative.
Enter value for report_name:
```

下面是上一个执行生成的报告的一小部分摘录（完整的报告请查看文件ashrpt_1_0212_2217.txt）。

- 关于报告的一般信息：

```
          Analysis Begin Time:  12-Feb-14 22:12:30
            Analysis End Time:  12-Feb-14 22:17:30
                 Elapsed Time:        5.0 (mins)
            Begin Data Source:  V$ACTIVE_SESSION_HISTORY
              End Data Source:  V$ACTIVE_SESSION_HISTORY
                 Sample Count:       4,583
      Average Active Sessions:       15.28
  Avg. Active Session per CPU:        3.82
                Report Target:  None specified
```

- top等待事件：

```
                                                    Avg Active
Event                     Event Class    % Event    Sessions
------------------------- ------------- -------- ----------
db file sequential read   User I/O          65.94      10.07
log file sync             Commit            14.42       2.20
CPU + Wait for            CPU CPU            5.59       0.85
write complete waits      Configuration      1.07       0.16
```

- 被引擎分解的活动：

```
                                                  Avg Active
Phase of Execution                 % Activity     Sessions
------------------------------ ---------- ----------
SQL Execution                       72.31       11.05
PLSQL Execution                      1.46        0.22
```

- top SQL语句和top wait event（这里只显示了前两个，报告里包含了5个）：

```
                                                         Sampled #
                   SQL ID             Planhash        of Executions     % Activity
----------------------- -------------------- -------------------- --------------
Event                          % Event Top Row Source                    % RwSrc
------------------------------ ------- --------------------------------- -------
          c13sma6rkr27c          569677903                  1005           21.99
db file sequential read          21.32 TABLE ACCESS - BY INDEX ROWID       15.64
SELECT PRODUCTS.PRODUCT_ID, PRODUCT_NAME, PRODUCT_DESCRIPTION, CATEGORY_ID, WEIG
HT_CLASS, WARRANTY_PERIOD, SUPPLIER_ID, PRODUCT_STATUS, LIST_PRICE, MIN_PRICE, C
ATALOG_URL, QUANTITY_ON_HAND FROM PRODUCTS, INVENTORIES WHERE PRODUCTS.CATEGORY_
ID = :B3 AND INVENTORIES.PRODUCT_ID = PRODUCTS.PRODUCT_ID AND INVENTORIES.WAREHO

          0yas01u2p9ch4                N/A                   382            8.34
db file sequential read           7.92 ** Row Source Not Available **       7.92
INSERT INTO ORDER_ITEMS(ORDER_ID, LINE_ITEM_ID, PRODUCT_ID, UNIT_PRICE, QUANTITY
) VALUES (:B4 , :B3 , :B2 , :B1 , 1)
```

4.2.8 SQL语句统计信息

在父级别与子级别上，分别可以通过v$sqlarea和v$sql视图获得与SQL语句关联的游标的信息。此外，父级别的性能统计数据也可以在v$sqlstats视图中查到。尽管v$sqlarea和v$sql视图提供了更多的信息（或者说更多的列），但使用v$sqlstats视图有两个明显的好处。首先，它会保留更多的数据，因此即便是从库缓存中交换出的游标也可能会在v$sqlstats视图中查询到。其次，访问v$sqlstats视图需要更少的资源。由于这些动态性能视图的列太多（例如，在10.2.0.5版本中v$sql视图有72列，在12.1.0.1版本中该视图有91列），这里不可能一一列举（更多信息请参考*Oracle Database Reference*手册）。以下是可以从v$sql视图中获取的与性能关系最密切的信息及该信息所在的列。

- 游标的标识（address、hash_value、sql_id和child_number）。
- 与游标关联的SQL语句的类型（command_type）和SQL语句的文本（sql_text中的前1000个字符和sql_fulltext中的全部文本）。
- 用于打开硬解析游标的会话的服务（service）、用于硬解析的schema（parsing_schema_name和parsing_schema_id）以及在硬解析期间已到位的会话属性（module和action）。
- 如果SQL语句是从PL/SQL执行的，那么该信息是，PL/SQL程序的ID和SQL语句所在的行号（program_id和program_line#）。
- 已发生的硬解析的数量（loads）、游标失效的次数（invalidations）、发生第一次和最后一次硬解析的时间（first_load_time和last_load_time）、存储的outline category的名（outline_category）、SQL profile（sql_profile）、SQL patch（sql_patch）、生成执行计划期间所使用的SQL plan baseline（sql_plan_baseline），以及与游标相关联的执行计划的散列值（plan_hash_value）。
- 已经完成的解析、执行和获取调用的数量（parse_calls、executions和fetches）以及已处理的行数（rows_processed）。对于查询来说，该信息是，已获取所有行的次数（end_of_fetch_count）。
- 用于处理的总DB time（elapsed_time），花在CPU上的时间（cpu_time）或用来等待属于应用、并发、集群和用户I/O等待级别的事件的时间（application_wait_time、concurrency_wait_time、cluster_wait_time和user_io_wait_time），以及PL/SQL引擎和Java虚拟机已完成的处理的数量

（plsql_exec_time和java_exec_time）。所有值的单位都是微秒。

- 已经完成的逻辑读、物理读、直接写和排序的数量（buffer_gets、disk_reads、direct_writes和sorts）。

4.2.9　实时监控

鉴于上一节介绍的动态性能视图只能提供关于游标的累积统计数据，实时监控能提供游标执行期间的信息。有两个重要的执行细节需要注意。第一，实时监控提供的是执行期间的信息。换句话说，你不需要等待执行结束就能看到想要的信息。第二，实时监控的信息是根据游标独立存放的。因此，即使游标已经被刷出库缓存，相关信息可能还可以访问到。在某种程度上，实时监控的目的和ASH很像。实际上，ASH提供的是活动会话状态的历史信息，而实时监控提供的是游标执行的历史信息。

注意，要使用实时监控，必须有Tuning Pack选件的许可。此外，实时监控只在11.1版本之后才开始支持。如果初始化参数control_management_pack_access没有设置成diagnostic+tuning，那么实时监控是无效的。

由于监控所有的执行是无意义的，因此数据库引擎会默认在以下三种情况下启用监控。

- CPU和磁盘I/O的时间总和超过了5秒的执行。
- 使用并行处理的执行。
- 通过指定monitor hint显式启用实时监控的SQL语句（同样可以使用no_monitor hint来显式禁用实时监控）。

警告　在以下两种情况下，数据库引擎会对特定执行自动禁用实时监控。第一，执行计划超过300行。第二，监控的数量超过了每颗CPU上20个并行执行。要想越过这些限制，你可以分别加大未公开的初始化参数_sqlmon_max_planlines和_sqlmon_max_plan的默认值。由于加大默认值会导致更高的CPU和内存消耗，在没有谨慎测试修改前，不要把它们设置成过高的值。

要查看当前监控的是哪个操作，可以直接查询v$sql_monitor视图或执行dbms_sqltune包下的report_sql_monitor_list函数。对于每个受监控的执行，数据库引擎都会提供基本信息，比如操作是否仍在执行，与受监控操作相关的SQL语句，以及像DB time使用率这样的关键性能指标。图4-10显示由Enterprise Manager提供的部分信息。

Status	Duration	SQL ID	User	Parallel	Database Time	IO Requests	Start
	30.0s	5kwfj03dc3dp	SOE		27.9s	9,162	10:59:17 AM
	10.1m	fhrwfq3tyqws	SOE	8	1.3h	1,548K	10:40:13 AM
	1.0s	fm1s40a43y26	SOE	8	7.1s	584	10:39:41 AM
	8.3m	1yxmh224tx4	SOE	8	33.9m	730K	10:30:40 AM
	17.7ms	dmdh8rpr6mxs	SOE		17.7ms	1	10:55:14 AM

图4-10　监控操作列表

要想查看所有实时监控获取到的信息，你需要使用dbms_sqltune包下的report_sql_monitor函数生成报告。这样的操作可以在任何能够执行SQL语句的工具里实现，在Enterprise Manager的几个页面里也可以实现（例如，Performance菜单下的SQL Monitoring连接）。report_sql_monitor函数包含若干输入变量并输出一个包含报告的CLOB。有些输入变量用来指定监控信息，还有的用来指定报告的格式和显示的数据。例如，sql_id参数用来指定需要显示的SQL语句信息（如果指定为NULL，那么会显示最后一次操作），type参数用来指定生成报告的格式（建议最好用active；text、html和xml同样也可以）。以下查询摘自report_sql_monitor.sql脚本，展示如何生成报告。

```
SELECT dbms_sqltune.report_sql_monitor(sql_id => '5kwfj03dc3dp1',
                                        type   => 'active')
FROM dual
```

在大多数情况下，报告包含了所有你需要理解的信息。对于一个活动报告来说（text和html格式的报告提供的信息较少），会提供如下信息。

- 执行的基本信息与关键性能指标的摘要（图4-11）

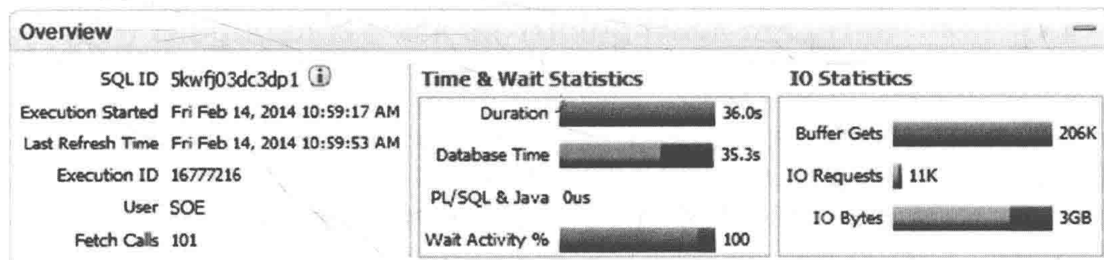


图4-11　监控操作的概述信息

- 执行计划，包含操作级别的性能指标（图4-12）

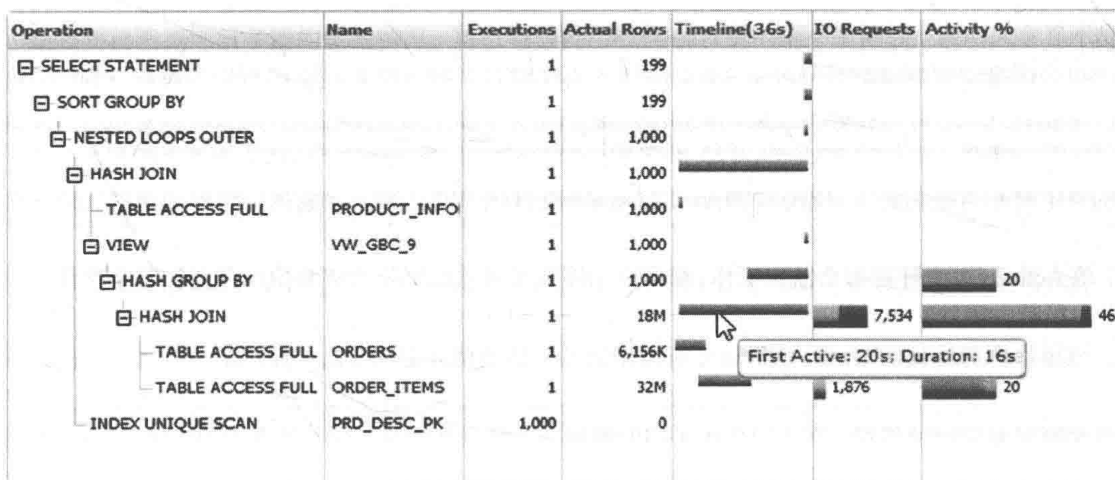


图4-12　监控操作的执行计划

❑ 显示CPU使用和执行期间经历的等待事件的活动示意图（图4-13）

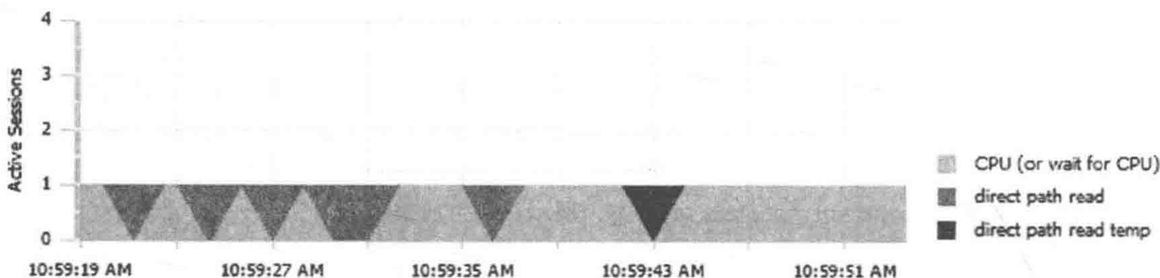


图4-13　监控操作的活动示意图

❑ 显示执行期间某些度量值的改变的若干图表（图4-14）

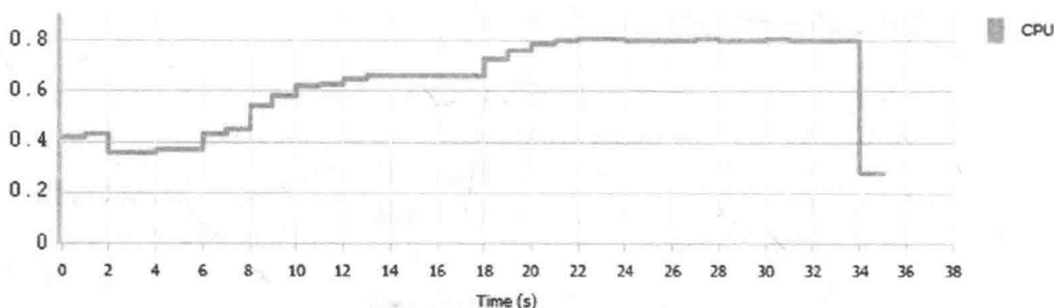


图4-14　监控操作的CPU使用率图表

要显示监控操作的相关SQL语句文本，需单击SQL ID旁边的（i）图标。之后会弹出窗口显示SQL语句文本。此外，如果操作里包含绑定变量，也会显示绑定变量名、位置、数据类型和绑定变量值。

报告中最有意思的部分是Plan Statistics表。有了它，不仅可以看到在操作级别执行计划的资源汇总，还能看到执行计划操作完成的时间和返回的行数。例如，在图4-12中，可以看到从执行开始20秒后HASH JOIN操作（鼠标高亮的那一行）返回了1800万行，并持续执行了16秒（你可以把鼠标停在时间线的条上显示这些信息）。使用Enterprise Manager查看报告时，另一个重要的特性是可以实时跟踪活动操作的执行。

在Activity表中针对每个监控操作，都有一个图表来显示CPU使用率和执行期间的等待事件。例如，图4-13显示，虽然执行开始时花在CPU上的大部分时间基本等同于直接读所用的时间，但在执行的最后，该操作全部仰仗CPU。注意只有并行操作活动会话的值才会大于1。

Metrics表显示如下若干性能指标的图表：CPU使用率、磁盘I/O请求数、磁盘I/O吞吐量、PGA使用量和临时表空间使用量。图4-14显示了CPU使用率。注意该图显示的内容早已在activity chart里看到过：随着执行的增加，CPU使用率上升。

Plan表显示包括由查询优化器估量的所有信息在内的执行计划。例如，你可以通过它查询到哪个执行计划操作用到了谓词推入。最后，针对并行操作，Parallel表显示涉及执行的每个进程的繁忙度。

复合数据库操作

在11.1版本中，实时监控只能监控SQL语句，因此有时也被叫作实时SQL监控。从11.2版本开始，此特性也开始支持PL/SQL块。最终，从12.1版本开始，使用复合数据库操作可以把数个SQL语句或PL/SQL块当作一个操作来处理。换句话说，可以把实时监控扩展成对业务有意义的用户自定义操作。例如，使用复合数据库操作来定义批处理任务执行的所有SQL语句都作为单个操作受到监控。

要定义复合数据库操作，你需要为监控的任务起名。有以下三种方法可以实现。

- Generic：使用`dbms_sql_monitor`包，具体说是`begin_operation`和`end_operation`函数来明确指定操作的开始和结束。这个方法可以在任何开发语言中使用。
- Java：使用`java.sql.Connection`接口的`setClientInfo`方法。这个技术只对JDBC 4.1及之后的版本有效。
- OCI：使用`OCIAttrSet`函数设置`OCI_ATTR_DBOP`会话属性。

4.3 使用 Diagnostics Pack 和 Tuning Pack 进行分析

要使用Diagnostics Pack做分析，建议使用Enterprise Manager的performance页（无论使用的是Database Control、Grid Control还是Cloud Control）。因此这一部分的结构和例子都基于Enterprise Manager。注意，该部分的所有例子也都适用于Cloud Control 12.1.0.3.0版本的Enterprise Manager页。万一你用不了Enterprise Manager，我也会介绍一些脚本，这些脚本使用动态性能视图来生成类似的信息，你可以使用SQL*Plus来执行相同的分析。强调一下，如果使用Enterprise Manager，分析会更简单一些。不使用Enterprise Manager的唯一优势是可以非常灵活方便地访问到所有可用数据。

4.3.1 数据库服务器负载

想要获得数据库服务器的负载情况，可以查看Performance首页的第一个图表。该图表会显示实时或历史数据。由于本章的目标是实时分析性能问题（或刚发生过的），这里默认使用实时数据。因此，一小时之内的数据都是可用的。注意图表显示的实时数据同样可以通过`v$metric_history`视图查到。

当监控数据库服务器负载情况时，不仅要查看数据库服务器是否是CPU bound（换句话说，是否所有的CPU内核都被充分利用），同时也要关注消耗大量CPU时间的进程是否与数据库实例有关。图4-15是一个CPU bound数据库服务器的例子，大约持续了5分钟。然而数据库实例的后台和前台进程经常只使用两个CPU内核。实际上，当数据库服务器是CPU bound时，大约有6个CPU内核会被不属于数据库实例的进程充分使用。因此在本例中，在五分钟内，有可能发生由于数据库服务器的其他进程造成数据库引擎经历性能问题。

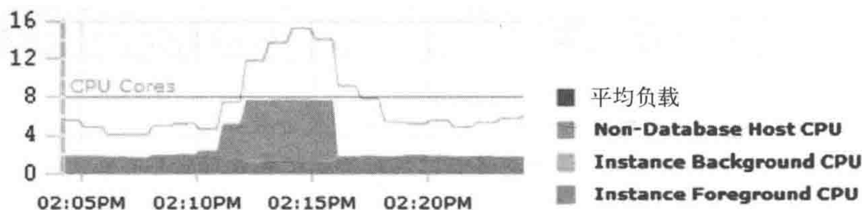


图4-15　Runnable Processes图表显示了数据库服务器级别的CPU使用率和平均负载

遗憾的是，直到11.1版本（包含11.1版本），Runnable Processes图表仅能显示平均负载（在Linux数据库服务器上，该值与/process/loadavg的值相同）。这是因为在之前的版本中相关值的度量不可用。

警告　当数据库服务器配置使用多线程并行处理的CPU时，初始化参数cpu_count和v$osstat视图的NUM_CPUS值都会被设成总线程数。因此，在Runnable Process图表里红线代表线程数，而不是CPU内核数（如上所述）。注意使用线程数来评估数据库是否为CPU bound会造成误导。实际上，100%使用所有线程是不可能的。同时注意虚拟化也有同样的问题。

如果无法使用Enterprise Manager，可以使用host_load_hist.sql脚本来显示与图4-15相同的信息。注意，脚本不需要参数。度量会显示一个小时内所有可用的数据。以下是脚本输出的一段节选：

```
SQL> @host_load_hist.sql

BEGIN_TIME DURATION DB_FG_CPU DB_BG_CPU NON_DB_CPU OS_LOAD NUM_CPU
---------- -------- --------- --------- ---------- ------- -------
14:05:00      60.10      1.71      0.03       0.03    4.09       8
14:06:00      60.08      1.62      0.03       0.04    4.13       8
14:07:00      59.10      1.89      0.03       0.04    4.96       8
14:08:00      60.11      1.93      0.03       0.03    5.29       8
14:09:00      60.09      1.73      0.03       0.59    4.60       8
14:10:00      60.10      1.57      0.02       3.64    7.50       8
14:11:00      60.16      1.15      0.02       6.60   11.82       8
14:12:00      60.11      1.21      0.02       6.60   13.77       8
14:13:00      60.28      1.17      0.02       6.62   15.30       8
14:14:00      59.24      1.19      0.02       6.55   14.06       8
14:15:00      60.09      1.59      0.04       0.18    9.19       8
14:16:00      60.09      1.77      0.03       0.03    7.88       8
14:17:00      60.09      1.72      0.03       0.04    5.45       8
14:18:00      60.11      1.87      0.03       0.03    5.28       8
14:19:00      60.09      1.77      0.03       0.03    5.54       8
14:20:00      60.08      1.72      0.03       0.04    4.83       8
```

4.3.2　系统级别分析

如果继续系统级别分析，应该从Top Activity页面开始。我不建议使用Performance主页，特别是Average Active Session图表，因为只有在单个等待级别消耗了大部分响应时间时，这个页面的挖掘功能才能快速定位性能问题。

提示 如果可以使用ASH Analytics，则应该用它代替Top Activity页面。在某些情况下，更灵活地选取时间段分析、添加过滤器以及根据众多维度汇总数据，可以获得更简单详细的分析。

Top Activity页面会显示实时和历史数据。由于本章的目标是实时分析性能问题（或刚发生过的），这里默认使用实时数据。因此，一小时之内的数据都是可用的。请注意，Top Activity页面显示的实时数据同样可以通过v$active_session_history视图查到。

Top Activity页面会提供以下三组数据。

- Activity图表（图4-16）显示最后一小时的数据。它还会把DB time分成CPU使用率和等待级别。

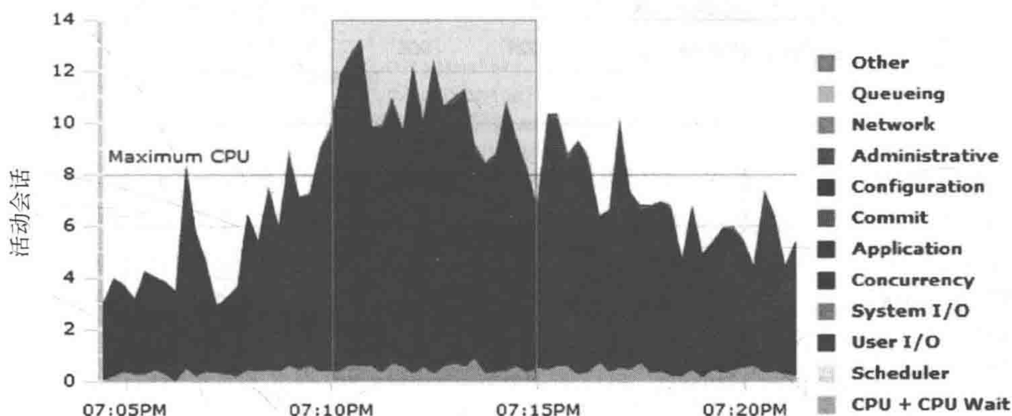


图4-16 Activity图表显示系统级别的CPU使用率和等待级别

- Top SQL表（图4-17），显示在activity图表中所选择的五分钟间隔里，最耗时的SQL语句。每条SQL语句会显示其总活动百分比以及SQL ID。

Select	Activity (%)	SQL ID	SQL Type
	24.56	c13sma6rkr27c	SELECT
	20.63	8dq0v1mjngj7t	SELECT
	12.45	7hk2m2702ua0g	SELECT
	11.95	bymb3ujkr3ubk	INSERT
	8.32	0yas01u2p9ch4	INSERT
	6.87	0bzhqhhj9mpaa	INSERT
	5.94	8z3542ffmp562	SELECT
	3.37	5mddt5kt45rg3	UPDATE
	2.44	f9u2k84v884y7	UPDATE
	.83	0w2qpuc6u2zsp	PL/SQL EXECUTE

图4-17 Top SQL表显示系统级别上最耗时的SQL语句

- Top Session表（图4-18），显示在activity图表中所选择的五分钟间隔里，最耗时的会话。每个会话会显示其总活动百分比以及会话信息（ID、用户名以及打开它的程序）。

活动 (%)	会话ID	用户名	程序
1.68	232	SOE	JDBC Thin Client
1.68	16	SOE	JDBC Thin Client
1.58	136	SOE	JDBC Thin Client
1.55	156	SOE	JDBC Thin Client
1.51	170	SOE	JDBC Thin Client
1.48	127	SOE	JDBC Thin Client
1.48	74	SOE	JDBC Thin Client
1.48	162	SOE	JDBC Thin Client
1.48	68	SOE	JDBC Thin Client
1.45	224	SOE	JDBC Thin Client

Total Sample Count: 3,103

图4-18　Top Sessions表显示最耗时的会话

Activity图表有两个目的。首先，它可以让你从数据库引擎的角度了解发生了什么。例如，通过图4-16，你不仅可以知道平均活动会话数为3~13（由于CPU数为8，因此数据库引擎属于中度负载），还可以知道用户I/O等待级别占用了大部分DB time。其次，你可以使用它来显示五分钟间隔内你想要查看的详细数据库负载信息。因此，如果你没有对某一特定时刻进行分析，通常可以选择活动会话数最多的那个时间段。换句话说，你应该选择负载最高的时间段。

如果你无法使用Enterprise Manager，可以使用ash_activity.sql脚本来显示最后一小时的活动率。这个脚本的目的是显示平均活动会话数和对应每个等待级别的百分比。下面的例子是基于图4-16显示的时间段产生的输出结果（注意，脚本跟着的两个参数all，代表输出结果没有限制到某个会话或某个SQL语句），与图4-16唯一不同的是数据以分钟为单位聚合。

```
SQL> @ash_activity.sql all all
TIME   AvgActSes   CPU% UsrIO% SysIO% Conc%  Appl% Commit% Config% Admin%   Net% Queue% Other%
----- --------- ------ ------ ------ ------ ------ ------- ------- ------ ------ ------ ------
19:04       3.8    6.2   93.8    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:05       3.6    8.0   92.0    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:06       5.6    4.8   95.2    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:07       3.4    7.9   92.1    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:08       6.0    5.0   92.5    0.0    2.5    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:09       7.5    6.9   93.1    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:10      11.2    3.7   96.3    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:11      10.4    4.5   95.5    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:12      10.9    2.9   97.1    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:13       9.8    6.5   93.5    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:14       9.2    3.8   96.2    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:15       8.6    5.2   94.8    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:16       8.0    4.6   95.4    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:17       7.6    5.1   94.9    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:18       6.1    4.4   95.6    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:19       5.7    5.0   95.0    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:20       6.0    7.2   92.8    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:21       4.8    4.5   95.5    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:22       4.9    5.7   94.3    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
```

一旦选择了你想要关注的五分钟间隔，就应该去Top SQL页面查看相应信息（图4-17）。如果显示少数SQL语句占用了大百分比的活动率（比如，单个SQL语句的活动率达到了两位数），就能定位到需要进一步分析的SQL语句。例如，根据图4-17，有7个查询占用了90%多的活动率。因此，为了降低系统负载，应该关注这些语句。

要想不使用Enterprise Manager显示图4-17的数据，可以使用ash_top_sqls.sql脚本。注意，该脚本需要三个输入参数。前两个参数用来指定显示数据的时间段（本例是开始和结束的时间戳）。第三个参数指定具体会话（这里指定all）。以下输出的数据与图4-17显示的相同。

```
SQL> @ash_top_sqls.sql 2014-02-04_19:10:02.174 2014-02-04_19:15:02.174 all

Activity% DB Time CPU% UsrIO% Wait% SQL Id        SQL Type
--------- ------- ----- ------ ----- ------------- --------------
     24.6     744   4.2   95.8   0.0 c13sma6rkr27c SELECT
     20.6     625   0.3   99.7   0.0 8dq0v1mjngj7t SELECT
     12.4     377   1.1   98.9   0.0 7hk2m2702ua0g SELECT
     12.0     362   1.9   98.1   0.0 bymb3ujkr3ubk INSERT
      8.3     252   3.6   96.4   0.0 0yas01u2p9ch4 INSERT
      6.9     208   1.4   98.6   0.0 0bzhqhhj9mpaa INSERT
      5.9     180   2.2   97.8   0.0 8z3542ffmp562 SELECT
      3.4     102   5.9   94.1   0.0 5mddt5kt45rg3 UPDATE
      2.4      74   2.7   97.3   0.0 f9u2k84v884y7 UPDATE
      0.8      25 100.0    0.0   0.0 0w2qpuc6u2zsp PL/SQL EXECUTE
```

如果没有突出的SQL语句，那么显然高活动率是由很多条SQL语句造成的。因此，这表明要想提升性能应该主要修改应用。遇到类似情况时，建议查看其他维度的活动率聚合。默认情况下，Top Activity页面显示Top Sessions表（图4-18）。但是在这张表顶部的下拉列表里，可以选择以其他维度对数据进行聚合，如Top Services、Top Modules、Top Actions（图4-19）和Top Clients。有时这对找到造成高负载的应用组件或客户端很有帮助。请注意，有一些维度只有在分析的应用正确设置了第2章介绍的会话属性后，才会提供有用的信息。

Activity (%)	Service	Module	Action
24.01	DBM11203.antognini.ch	New Order	getProductDetailsByCategory
23.78	DBM11203.antognini.ch	New Order	
14.57	DBM11203.antognini.ch	Process Orders	
10.22	DBM11203.antognini.ch	Browse Products	getCustomerDetails
8.35	DBM11203.antognini.ch	New Order	getCustomerDetails
6.80	DBM11203.antognini.ch	New Customer	
5.87	DBM11203.antognini.ch	New Order	getProductQuantity
1.64	SYS$BACKGROUND		
1.61	DBM11203.antognini.ch	Browse and Update Orders	getCustomerDetails
.77	DBM11203.antognini.ch		

Total Sample Count: 3,103

图4-19　Top Actions表显示根据service、module和action聚合的最耗时组件信息

注意　Enterprise Manager显示的Total Sample Count值代表创建图表时使用的历史活动会话取样数。例如，图4-18使用了3103个取样。

当考虑到像Top Sessions或Top Actions表那样显示活动率时，同样需要查找占用大量活动率的组件。例如，虽然根据图4-18没有哪个会话的活动率超过2%，但图4-19显示少数modle/action造成了大部分负载，因此除了Top SQL表指出的部分外，你或许也应该检查一下这些module/action。

要想不使用Enterprise Manager来根据特定维度显示数据聚合，可以使用如下脚本中的一个：ash_top_sessions.sql、ash_top_services.sql、ash_top_modules.sql、ash_top_actions.sql、ash_top_clients.sql、ash_top_files.sql、ash_top_objects.sql和ash_top_plsql.sql。维度信息已经明确写在脚本名里。注意所有脚本都需要两个输入参数（开始时间戳和结束时间戳）来指定显示数据的时间段。以下两个例子显示的输出是由ash_top_sessions.sql和ash_top_actions.sql脚本生成的，分别等同于图4-18和图4-19：

```
SQL> @ash_top_sessions.sql 2014-02-04_19:10:02.174 2014-02-04_19:15:02.174

Activity% DB Time CPU% UsrIO% Wait% Session Id User Name Program
--------- ------- ---- ------ ----- ---------- --------- ----------------
      1.7      52  9.6   90.4   0.0        232 SOE       JDBC Thin Client
      1.7      52  3.8   96.2   0.0         16 SOE       JDBC Thin Client
      1.6      49  4.1   95.9   0.0        136 SOE       JDBC Thin Client
      1.5      48  6.3   93.8   0.0        156 SOE       JDBC Thin Client
      1.5      47  4.3   93.6   2.1        170 SOE       JDBC Thin Client
      1.5      46 10.9   89.1   0.0        127 SOE       JDBC Thin Client
      1.5      46  6.5   93.5   0.0         74 SOE       JDBC Thin Client
      1.5      46  8.7   89.1   2.2        162 SOE       JDBC Thin Client
      1.5      46  2.2   97.8   0.0         68 SOE       JDBC Thin Client
      1.5      45  8.9   91.1   0.0         77 SOE       JDBC Thin Client

SQL> @ash_top_actions.sql 2014-02-04_19:10:02.174 2014-02-04_19:15:02.174

Activity% DB Time CPU% UsrIO% Wait% Module                   Action
--------- ------- ---- ------ ----- ------------------------ --------------------------
     24.0     745  4.3   95.7   0.0 New Order                getProductDetailsByCategory
     23.8     738  6.0   94.0   0.0 New Order
     14.6     452  1.5   98.5   0.0 Process Orders
     10.2     317  0.0  100.0   0.0 Browse Products          getCustomerDetails
      8.3     259  1.2   98.8   0.0 New Order                getCustomerDetails
      6.8     211  2.8   97.2   0.0 New Customer
      5.9     182  3.3   96.7   0.0 New Order                getProductQuantity
      1.6      51 37.3    3.9  58.8
      1.6      50  0.0  100.0   0.0 Browse and Update Orders getCustomerDetails
      0.8      24 45.8    0.0  54.2
```

4.3.3　会话级别分析

如果执行会话级别分析，出发点应该基于会话是否还存在。如果会话存在，则可以在Performance菜单下的Search Sessions菜单中搜索到会话信息。另外，也可以在Top Activity页面，特别是Top Sessions

表中找到对应的Session ID链接。

会话级别的活动率页面提供了以下三组数据。

- ❑ Activity图表（图4-20）显示CPU与等待事件分别占用的DB time。它会显示一个小时的数据。如果活动率为0，代表会话是空闲状态。如果活动率达到100%，则代表会话完全忙于处理用户调用。

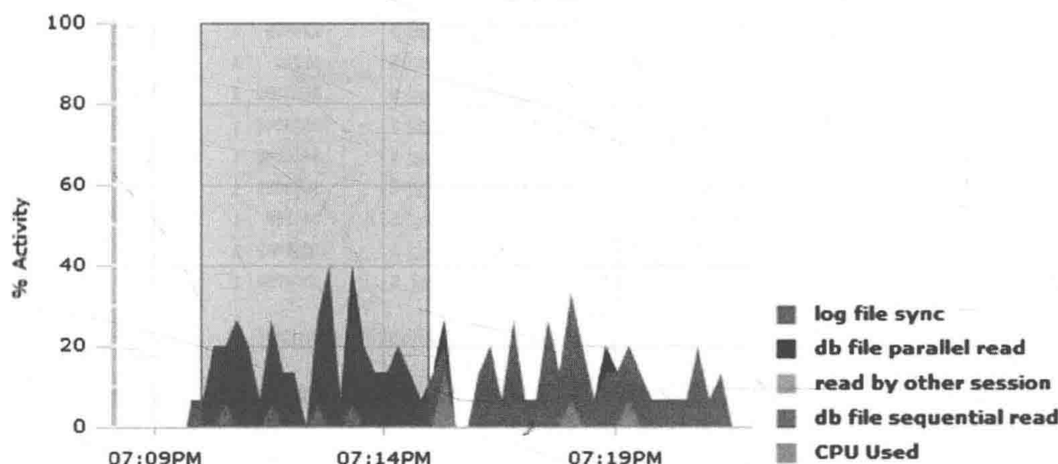


图4-20　会话级别活动率图表显示单独会话的CPU使用率和等待事件

- ❑ Active session history aggregated data（图4-21）显示了在activity图表中所选择的5分钟间隔里最耗时的SQL语句。对于每条语句，会给出总活动率、SQL ID、执行计划的散列值和一些会话属性。

Activity (%) ▼	SQL ID	SQL Command	Plan Hash Value	Module
23.08	c13sma6rkr27c	SELECT	2583456710	New Order
13.46	bymb3ujkr3ubk	INSERT	494735477	New Order
13.46	8dq0v1mjngj7t	SELECT	900611645	New Order
11.54	0yas01u2p9ch4	INSERT	0	New Order
9.62	8dq0v1mjngj7t	SELECT	900611645	Browse Products
5.77	8z3542ffmp562	SELECT	1655552467	New Order
3.85	0ruh367af7gbw	SELECT	3322340634	Browse and Update Orders
3.85	f9u2k84v884y7	UPDATE	1628223527	Process Orders
3.85	7hk2m2702ua0g	SELECT	1278617784	Process Orders
3.85	0bzhqhhj9mpaa	INSERT	0	New Customer

图4-21　Active session history aggregated data显示会话级别上最耗时的SQL语句

- ❑ Active session history raw data（图4-22）显示了在activity图表中所选择的五分钟间隔里，取样的详细信息。

Sample Time ▼	SQL ID	SQL Type	Plan Hash Value	Wait Event	P1 Value	P2 Value	P3 Value	Time Waited (mhu s)
2/4/14 7:14:54 PM	0y1prvxqc2ra9	SELECT	302912750	CPU				
2/4/14 7:14:45 PM	0bzhqhhj9mpaa	INSERT	0	db file sequential read	5	3652838	1	14859
2/4/14 7:14:44 PM	8z3542ffmp562	SELECT	1655552467	db file sequential read	5	1084446	1	3923
2/4/14 7:14:24 PM	7hk2m2702ua0g	SELECT	1278617784	db file sequential read	5	3426419	1	12010
2/4/14 7:14:22 PM	8z3542ffmp562	SELECT	1655552467	db file sequential read	5	1087383	1	7257
2/4/14 7:14:16 PM	bymb3ujkr3ubk	INSERT	494735477	db file sequential read	5	3427835	1	15603
2/4/14 7:14:03 PM	bymb3ujkr3ubk	INSERT	494735477	db file sequential read	5	234809	1	34997
2/4/14 7:14:01 PM	7hk2m2702ua0g	SELECT	1278617784	db file sequential read	5	10221	1	33044
2/4/14 7:14:00 PM	0yas01u2p9ch4	INSERT	0	db file sequential read	5	3576201	1	78505
2/4/14 7:13:43 PM	0yas01u2p9ch4	INSERT	0	db file sequential read	5	3515548	1	6447
2/4/14 7:13:42 PM	5mddt5kt45rg3	UPDATE	1628223527	db file sequential read	5	3419246	1	6663
2/4/14 7:13:36 PM	0bzhqhhj9mpaa	INSERT	0	db file sequential read	5	3653055	1	6129
2/4/14 7:13:35 PM	8dq0v1mjngj7t	SELECT	900611645	db file sequential read	5	346169	1	20058
2/4/14 7:13:22 PM	c13sma6rkr27c	SELECT	2583456710	db file sequential read	5	1088402	1	7206
2/4/14 7:13:20 PM	c13sma6rkr27c	SELECT	2583456710	db file sequential read	5	1078754	1	29743

图4-22 Active session history raw data显示取样的详细信息

会话级别分析与上一节介绍的系统级别分析很像。唯一的主要区别是，在会话级别，Enterprise Manager没有根据多个维度聚合数据的选项。你关注的是单独的会话，而且在大多数情况下，只与top SQL语句有关。

如果不能访问Enterprise Manager，可以分别使用ash_activity.sql和ash_top_sqls.sql脚本来显示等同于图4-20和图4-21的数据。请参考上一节中对脚本的简介。要使用它们，只需要指定要分析的会话ID即可。可以查询v$active_session_history视图来显示图4-22中的数据。下面举例说明如何使用ash_activity.sql脚本来显示与图4-20相似的数据（注意，第一个参数指定会话ID）。

```
SQL> @ash_activity.sql 232 all

TIME  AvgActSes   CPU% UsrIO% SysIO% Conc%  Appl% Commit% Config% Admin%   Net% Queue% Other%
----- --------- ------ ------ ------ ------ ------ ------- ------- ------ ------ ------ ------
19:10       0.2   11.1   88.9    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:11       0.2    8.3   91.7    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:12       0.1    0.0  100.0    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:13       0.2    7.1   92.9    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:14       0.2    0.0  100.0    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:15       0.1   33.3   66.7    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:16       0.1    0.0  100.0    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:17       0.2    0.0  100.0    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:18       0.2    0.0  100.0    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:19       0.2   10.0   90.0    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
19:20       0.1    0.0  100.0    0.0    0.0    0.0     0.0     0.0    0.0    0.0    0.0    0.0
```

4.3.4 SQL语句信息

当你关注某个特定SQL语句时，可以通过以下两种方式显示关于该SQL语句的详细信息：在显示排名靠前的SQL语句的其中一个表中单击该SQL语句的SQL ID（例如，图4-17和图4-21）；或在

Performance菜单中通过Search SQL链接进行搜索。这样你会进入相关的SQL Detail页面。注意当存在多个执行计划时，可以在SQL语句文本和标签之间的Plan Hash Value下拉列表中选择其中一个。除了SQL语句，SQL Detail页面还提供如下标签。

- Statistics标签显示执行该SQL语句的平均活动会话数（图4-23）、执行统计信息（图4-24）以及与该SQL语句关联的游标的相关信息。注意执行统计信息是从游标在库缓存中初始化时开始计算的累计值。该信息只有在游标还没有从库缓存中超期时才有效。

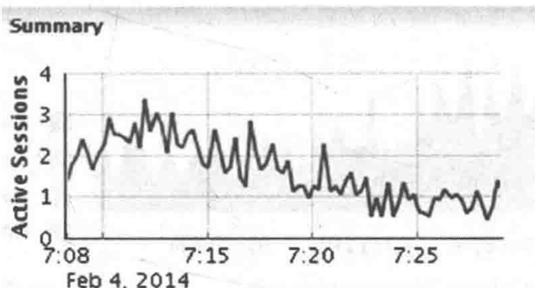
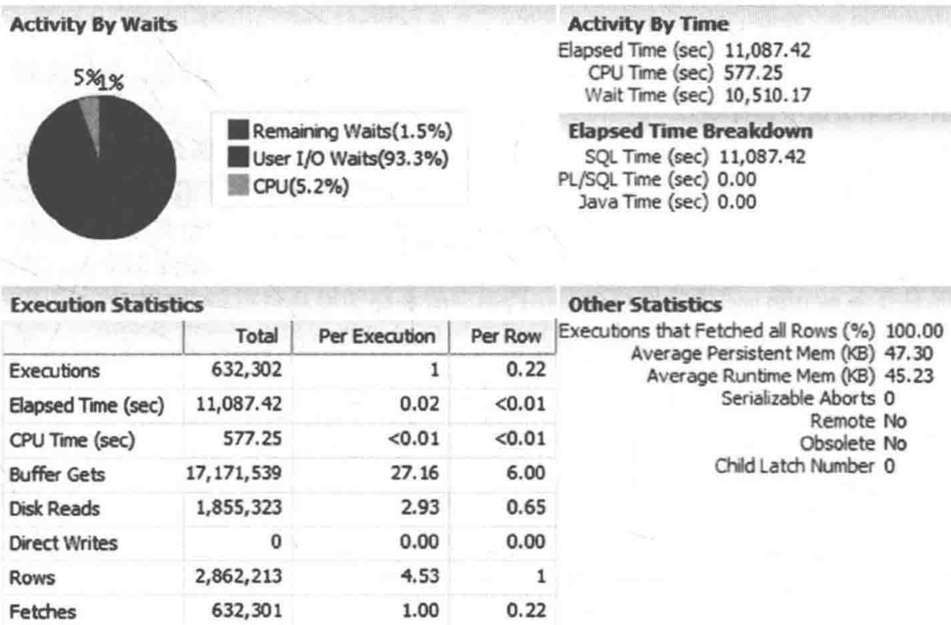


图4-23 Statistics标签的汇总图表显示与单个SQL语句相关的平均活动会话



Execution Statistics

	Total	Per Execution	Per Row
Executions	632,302	1	0.22
Elapsed Time (sec)	11,087.42	0.02	<0.01
CPU Time (sec)	577.25	<0.01	<0.01
Buffer Gets	17,171,539	27.16	6.00
Disk Reads	1,855,323	2.93	0.65
Direct Writes	0	0.00	0.00
Rows	2,862,213	4.53	1
Fetches	632,301	1.00	0.22

图4-24 Statistics标签的执行统计信息显示单个SQL语句的运行时行为信息

- Activity标签（图4-25）显示CPU使用率和等待事件占用的DB time。这里显示一个小时的数据。
- Plan标签显示与SQL语句相关的执行计划。该信息只有在执行计划还没有从共享池里超期时才会显示。第10章将会详细介绍如何阅读执行计划和判断它是否高效。

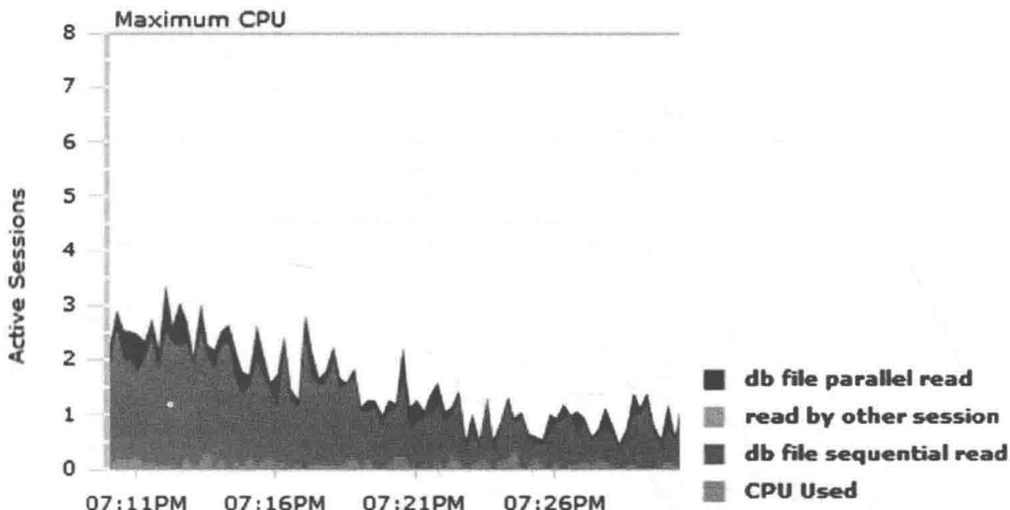


图4-25　SQL语句级别的活动率图表显示与单个SQL语句相关的CPU使用率和等待事件

- Plan Control标签显示与SQL语句相关的对象，如SQL profile和SQL plan baseline。这些对象会在第11章介绍。
- Tuning History标签显示由SQL Tuning advisor（详见第11章）生成的信息。
- SQL Monitoring标签，只从11.1版本起才有，显示实时监控的相关信息。如果这部分信息不可用，则无法选中该标签。

如果使用的是11.2及以后的版本，并且拥有Tuning Pack选件的授权，那么也可以从SQL Detail页面生成一份SQL Details Active报告。由于报告并没有比SQL Detail页面提供更多的信息，因此只有在想要把看到的信息保存为HTML文件时才使用报告。这样可以在以后查阅或发给其他人。如有需要，不用Enterprise Manager也能生成同样的报告。可以使用dbms_sqltune包下的report_sql_detail函数。该函数需要几个输入参数并返回一个包含报告的CLOB。有些输入参数可以用来改变报告要显示的内容，而使用sql_id参数可以指定报告要显示的SQL语句。以下查询出自report_sql_detail.sql脚本，演示了如何生成一份报告：

```
SELECT dbms_sqltune.report_sql_detail(sql_id => 'c13sma6rkr27c')
FROM dual
```

如果不使用Enterprise Manager来显示执行统计信息（图4-24）和SQL语句的一般信息，可以使用如下脚本中的一个：sqlarea.sql、sql.sql和sqlstats.sql。顾名思义，它们分别从v$sqlarea、v$sql和v$sqlstats中提取数据。更多信息请参考4.4.4节。

提示　sqlarea.sql、sql.sql和sqlstats.sql提供了一个Enterprise Manager所没有的特性。它们不仅可以与Enterprise Manager一样，显示自从游标加载后的累积统计信息，还会记录最后n秒的统计信息。这对了解当前执行的统计信息很有用。实际上，对于在库缓存里停留很长时间的游标，由累积统计信息提供的信息或许会造成误导。

如果不使用Enterprise Manager显示单独SQL语句的活动率，可以使用ash_activity.sql脚本，并指定第一个参数为all（这代表在会话级别没有限制），第二个参数指定为SQL语句的ID。下面的例子与图4-25类似。

```
SQL> @ash_activity.sql all c13sma6rkr27c

TIME AvgActSes CPU% UsrIO% SysIO% Conc% Appl% Commit% Config% Admin% Net% Queue% Other%
----- --------- ---- ------ ------ ----- ----- ------- ------- ------ ---- ------ ------
19:10       2.6  3.2   96.8    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
19:11       2.4  5.5   94.5    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
19:12       2.9  2.3   97.7    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
19:13       2.4  5.6   94.4    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
19:14       2.4  4.2   95.8    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
19:15       2.1  6.3   93.7    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
19:16       1.8  4.7   95.3    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
19:17       2.1  1.6   98.4    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
19:18       1.9  5.4   94.6    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
19:19       1.2  6.8   93.2    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
19:20       1.5  8.9   91.1    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
19:21       1.2  2.7   97.3    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
19:22       1.2  8.2   91.8    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
19:23       0.8 10.2   89.8    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
19:24       1.0 15.3   84.7    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
19:25       0.8 11.1   88.9    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
19:26       1.0  6.6   93.4    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
19:27       0.9  9.3   90.7    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
19:28       0.8 10.0   90.0    0.0   0.0   0.0     0.0     0.0    0.0  0.0    0.0    0.0
```

4.4 不使用 Diagnostics Pack 进行分析

不使用Diagnostics Pack选件来分析性能问题的主要操作同上一节介绍的基本一致。很显然，执行分析时不使用Diagnostics Pack选件授权的动态性能视图。这里有两个要求：第一，不能使用Enterprise Manager；第二，你能使用的大多数动态性能视图只提供累积的统计信息。特别是并不存在活动会话历史的替代品。结果就是你无法查看之前的几分钟发生了什么，也无法查询会话的历史操作。唯一没有提供累积统计信息的动态视图保存着度量值。但由于度量值主要关注比率和计数，这对分析性能问题没什么帮助。总之，分析就只能使用那些提供累积统计信息的动态性能视图了。

要想使用动态性能视图有效地分析性能问题，需要借助工具对视图提供的信息进行取样。这样的工具可以是简单的脚本或像Enterprise Manager一样复杂的工具。即使很多第三方工具提供了与Enterprise Manager类似的特性，但在这部分我们主要关注一组可以自由使用的脚本，这样就能适用于所有平台。由于大部分脚本都处理累积统计信息，它们的主要目的是要找出一小段时间内某统计值的变化率。因此脚本会反复选择同一个动态性能视图并计算每次选择间的差量。

4.4.1 数据库服务器负载

要评估数据库的负载，你无法使用历史度量值（需要Diagnostics Pack选件才可以使用）。应该查询v$metric视图来获取当前度量值。我使用host_load.sql脚本来获取当前值。该脚本有一个参数，用

来指定显示数据库服务器加载的分钟数。下面是一段脚本的输出（注意，数据与图4-15一致）：

```
SQL> @host_load.sql 16

BEGIN_TIME DURATION DB_FG_CPU DB_BG_CPU NON_DB_CPU OS_LOAD NUM_CPU
---------- -------- --------- --------- ---------- ------- -------
14:05:00      60.10      1.71      0.03       0.03    4.09       8
14:06:00      60.08      1.62      0.03       0.04    4.13       8
14:07:00      59.10      1.89      0.03       0.04    4.96       8
14:08:00      60.11      1.93      0.03       0.03    5.29       8
14:09:00      60.09      1.73      0.03       0.59    4.60       8
14:10:00      60.10      1.57      0.02       3.64    7.50       8
14:11:00      60.16      1.15      0.02       6.60   11.82       8
14:12:00      60.11      1.21      0.02       6.60   13.77       8
14:13:00      60.28      1.17      0.02       6.62   15.30       8
14:14:00      59.24      1.19      0.02       6.55   14.06       8
14:15:00      60.09      1.59      0.04       0.18    9.19       8
14:16:00      60.09      1.77      0.03       0.03    7.88       8
14:17:00      60.09      1.72      0.03       0.04    5.45       8
14:18:00      60.11      1.87      0.03       0.03    5.28       8
14:19:00      60.09      1.77      0.03       0.03    5.54       8
14:20:00      60.08      1.72      0.03       0.04    4.83       8
```

4.4.2 系统级别分析

在系统级别上进行分析时，首先需要检查整个系统的统计信息来确认数据库实例的负载情况。可以在v$system_wait_class视图中查询到这些统计信息。更确切地说，应该使用脚本（或工具）来对v$system_wait_class视图进行取样，以获得与图4-16类似的信息，也就是平均活动会话数和花费在每个等待级别的时间。下面是使用system_activity_sql脚本的例子，你需要指定以下两个参数。

- 第一个参数指定取样时间间隔。由于数据库引擎不会实时更新统计信息，将取样时间间隔指定为少于10~15秒通常没有意义。
- 第二个参数是样本的数量。

下面的例子是使用system_activity_sql脚本生成的输出，参数指定了20个间隔为15秒的样本（注意数据与图4-16显示的内容一致）。

```
SQL> @system_activity.sql 15 20

Time     AvgActSess Other% Net% Adm% Conf% Comm% Appl% Conc% SysIO% UsrIO% Sched%  CPU%
-------- ---------- ------ ---- ----  ----- ----- ----- ------ ------ ------ ------ ------
19:10:11        9.7    0.0  0.0  0.0    0.0   0.4   0.0    0.0    0.9   94.8    0.0   3.8
19:10:26       10.0    0.0  0.0  0.0    0.0   0.5   0.0    0.0    1.0   94.6    0.0   3.9
19:10:41       10.0    0.0  0.0  0.0    0.0   0.4   0.0    0.0    1.0   94.8    0.0   3.8
19:10:56        9.9    0.0  0.0  0.0    0.0   0.4   0.0    0.0    1.0   94.6    0.0   4.0
19:11:11        9.8    0.0  0.0  0.0    0.2   1.0   0.0    0.0    1.2   93.7    0.0   4.0
19:11:26        9.5    0.0  0.0  0.0    0.0   0.4   0.0    0.0    0.9   94.8    0.0   3.9
19:11:41        9.6    0.0  0.0  0.0    0.0   0.4   0.0    0.0    0.9   94.8    0.0   3.8
19:11:56        9.8    0.0  0.0  0.0    0.0   0.5   0.0    0.0    1.0   94.6    0.0   3.9
19:12:11        9.7    0.0  0.0  0.0    0.0   0.3   0.0    0.0    0.8   94.8    0.0   4.1
19:12:26        9.5    0.0  0.0  0.0    0.0   0.4   0.0    0.0    1.0   94.5    0.0   4.0
19:12:42        9.9    0.0  0.0  0.0    0.0   0.4   0.0    0.0    0.9   95.1    0.0   3.6
```

```
19:12:57         9.8   0.0  0.0  0.0    0.9  0.4  0.0    0.1    0.9  93.7   0.0   3.9
19:13:12         9.4   0.0  0.0  0.0    0.0  0.4  0.0    0.0    1.0  94.7   0.0   4.0
19:13:27         9.7   0.0  0.0  0.0    0.0  0.4  0.0    0.0    0.9  94.7   0.0   4.0
19:13:42         9.8   0.0  0.0  0.0    0.0  0.4  0.0    0.0    1.1  94.6   0.0   3.9
19:13:57        10.1   0.0  0.0  0.0    0.0  0.4  0.0    0.0    1.0  94.9   0.0   3.7
19:14:12         9.9   0.0  0.0  0.0    0.0  0.4  0.0    0.0    0.9  94.9   0.0   3.7
19:14:27         9.6   0.0  0.0  0.0    0.0  0.4  0.0    0.0    1.0  94.5   0.0   4.0
19:14:42         9.6   0.0  0.0  0.0    0.7  0.4  0.0    0.0    0.9  94.0   0.0   4.0
19:14:57         9.8   0.0  0.0  0.0    0.0  0.4  0.0    0.0    0.8  94.8   0.0   4.0
```

另一个用来检查全系统负载情况的工具是时间模型统计信息，特别是v$sys_time_model视图里的数据。它可以告诉你哪个引擎处理数据最多，万一处理数据最多的是SQL引擎，它也同样会告知你何种操作会影响性能，比如解析。同样在这里，我建议你使用脚本（或工具）来对动态性能视图的内容取样。例如，在一段时间内，time_model.sql脚本显示了一段时间内所有时间模型统计信息的详细信息。要使用它必须指定以下两个参数。

- 第一个参数指定取样时间间隔。由于数据库引擎不会实时更新统计信息，将取样时间间隔指定为少于10~15秒通常没有意义。
- 第二个参数是样本的数量。

下面的例子是使用time_model.sql脚本生成的输出，参数指定了2个间隔为15秒的样本（注意脚本只显示取样间隔期间统计值的变化）。

```
SQL> @time_model.sql 15 2

Time     Statistic                                         AvgActSess Activity%
-------- -------------------------------------------------- ---------- ---------
19:14:49 DB time                                                   9.8      98.6
         .DB CPU                                                   0.3       3.4
         .sql execute elapsed time                                 9.7      97.3
         .PL/SQL execution elapsed time                            0.1       1.2
         background elapsed time                                   0.1       1.4
         .background cpu time                                      0.0       0.4

Time     Statistic                                         AvgActSess Activity%
-------- -------------------------------------------------- ---------- ---------
19:15:04 DB time                                                   9.8      98.8
         .DB CPU                                                   0.3       3.5
         .sql execute elapsed time                                 9.7      97.8
         .parse time elapsed                                       0.0       0.3
         ..hard parse elapsed time                                 0.0       0.3
         .PL/SQL execution elapsed time                            0.1       1.2
         background elapsed time                                   0.1       1.2
         .background cpu time                                      0.0       0.3
```

你同样可以使用时间模块统计信息来确认观察到的大部分活动是否由某些会话产生。为此，你需要v$sess_time_model视图提供的会话级别统计信息。同样，应该使用脚本（或工具）来对动态性能视图的内容取样。我将基于active_sessions.sql脚本举例。脚本的目的是显示在给定的时间段里，top session花费了多少DB time。要使用此脚本，需要指定以下三个参数。

- 第一个参数指定取样时间间隔。由于数据库引擎不会实时更新统计信息，将取样时间间隔指定为少于10~15秒通常没有意义。

- 第二个参数是样本的数量。
- 第三个参数指定输出的会话数。通常指定少于10~20个会话是无意义的。

下面的例子是使用active_sessions.sql脚本生成的输出，参数指定了取样间隔为15秒，收集10个会话的信息（注意，数据与图4-18显示的内容一致）。

```
SQL> @active_sessions.sql 15 1 10

Time     #Sessions #Logins SessionId       Username            Program          Activity%
-------- --------- ------- --------------- ------------------- ---------------- ---------
19:14:49       117       0 195             SOE                 JDBC Thin Client       1.8
                           224             SOE                 JDBC Thin Client       1.5
                           225             SOE                 JDBC Thin Client       1.5
                           232             SOE                 JDBC Thin Client       1.5
                           7               SOE                 JDBC Thin Client       1.5
                           227             SOE                 JDBC Thin Client       1.4
                           74              SOE                 JDBC Thin Client       1.4
                           16              SOE                 JDBC Thin Client       1.4
                           171             SOE                 JDBC Thin Client       1.4
                           68              SOE                 JDBC Thin Client       1.4
                           Top-10 Total                                               14.9
```

注意，之前的输出也显示了每个间隔打开的会话和登录数。这个信息很重要，因为脚本无法发现在取样间隔期间终止的会话执行了哪些动作。所以，当你发现会话数在减少，或者登录数很高但会话数却没有按比例增加时，就应该提高警惕了。

根据脚本输出，如果大部分的活动率是由几个会话造成的（比如，单个会话的活动率至少是两位数百分比），你就应该定位会话以对其进行进一步的分析。根据上面的例子，如果没有突出的会话，就表明会话活动率很平均。因此，或许应该根据会话ID以外的维度对性能统计数据进行聚合。建议你使用Tanel Põder开发的脚本来实现。脚本名叫Snapper[①]（snapper.sql）。它的主要功能是以跟采样周期成反比的频率对v$session视图进行取样。取样期间Snapper会检查指定会话的状态，而对于活动会话，会收集它们活动率的信息（比如在执行的SQL语句）。由于Snapper是一个非常灵活且强大的脚本，可以使用很多参数，因此这里无法进行完整的介绍。这里只介绍一些基础知识并展示几个例子。有关更多信息，请阅读脚本的标头。

Snapper需要四个参数。

- 第一个参数指定需要取样的动态性能视图。如果指定常量ash，则会对v$session进行取样。这样做的目的是收集与活动会话历史相似的数据。指定这个参数后，也可以指定v$session视图的相关列来进行聚合。例如，ash=username+sql_id表示数据会根据v$session视图的username和sql_id列进行聚合（视图中的任何列都可以指定）。当指定常量stats时，会对v$sesstat、v$sess_time_model和v$session_event进行取样。
- 第二个参数指定取样周期，单位为秒。
- 第三个参数指定样本数量。
- 第四个参数指定取样的会话。这里可以指定单独会话ID、多个会话ID列表（用逗号分隔），指定常量all对所有会话进行取样，也可以指定查询返回的会话ID，或可用表达式的其中一个（例

① 可以在http://blog.tanelpoder.com/files/scripts.snapper.sql下载到该脚本。

如，user=chris表示查询由指定用户打开的所有会话。要获取可用表达式的完整列表，请参阅脚本的标头）。

第一个使用Snapper的例子显示如何收集与图4-17相似的信息。四个参数如下。

- 第一个参数（ash=sql_id）指定查询v$session视图，并根据SQL_ID聚合结果数据。
- 第二个参数（15）指定使用15秒取样间隔。
- 第三个参数（1）指定一个样本。
- 第四个参数（all）指定对所有会话进行取样。

```
SQL> @snapper.sql ash=sql_id 15 1 all

-------------------------
Active% | SQL_ID
-------------------------
   196% | c13sma6rkr27c
   186% | 8dq0v1mjngj7t
   122% | bymb3ujkr3ubk
   107% | 7hk2m2702ua0g
    82% | 0yas01u2p9ch4
    63% | 8z3542ffmp562
    62% | 0bzhqhhj9mpaa
    30% | 5mddt5kt45rg3
    26% |
    26% | f9u2k84v884y7
```

请注意，在上面的例子中，Active%列或许会大于100%。这在对多个会话进行取样时会发生。例如上面的输出，在取样期间，top SQL语句（c13sma6rkr27c）平均被1.96个会话执行。

第二个例子显示如何收集与图4-19相似的信息。对比上一个例子，只有第一个参数需要修改。它需要根据会话属性module和action聚合数据（ash=module+action）。

```
SQL> @snapper.sql ash=module+action 15 1 all
-----------------------------------------------------------------
Active% | MODULE                   | ACTION
-----------------------------------------------------------------
    97% | New Order                | getProductDetailsByCatego
    94% | New Order                |
    86% | Process Orders           |
    58% | Browse Products          | getCustomerDetails
    32% | New Order                | getCustomerDetails
    28% | New Customer             |
    22% | New Order                | getProductQuantity
     9% |                          |
     8% | Browse and Update Orders | getCustomerDetails
     3% | Browse Products          | getProductDetails
```

4.4.3 会话级别分析

前面使用Snapper的例子展示了如何分析整个系统的活动率（换句话说，所有会话的活动率）。然而Snapper同样可以只针对某个会话。对此，第四个参数就要由all改成具体的某个会话的ID。下面的两个例子分别展示了如何获得与图4-20和图4-21类似的信息。

```
SQL> @snapper.sql ash=event 15 1 172

----------------------------------------
Active% | EVENT
----------------------------------------
    22% | db file sequential read
     1% | ON CPU
     1% | db file parallel read

SQL> @snapper.sql ash=sql_id+module+action 15 1 172

--------------------------------------------------------------------------------
Active% | SQL_ID          | MODULE                     | ACTION
--------------------------------------------------------------------------------
     7% | c13sma6rkr27c | New Order                  | getProductDetailsByCatego
     3% | 8dq0v1mjngj7t | New Order                  | getCustomerDetails
     3% | 0yas01u2p9ch4 | New Order                  |
     1% | 7hk2m2702ua0g | Process Orders             |
     1% | 8dq0v1mjngj7t | Browse Products            | getCustomerDetails
     1% | 8dq0v1mjngj7t | Browse and Update Orders | getCustomerDetails
     1% | bymb3ujkr3ubk | New Order                  |
     1% | 8z3542ffmp562 | New Order                  | getProductQuantity
     1% | 0bzhqhhj9mpaa | New Customer               |
```

4.4.4 SQL语句信息

定位了大活动率的SQL语句之后，可以使用以下脚本中的一个来显示信息：sqlarea.sql、sql.sql和sqlstats.sql。顾名思义，它们分别从v$sqlarea、v$sql和v$sqlstats中提取数据。这三个脚本需要两个输入参数。

- 第一个参数指定SQL语句的ID。
- 第二个参数指定脚本显示的是从游标加载进库缓存后的累积统计值还是当前增加的统计值。将该参数设置成大于0的数字时，将启用后一种模式。那样的话，会根据参数（秒数）指定的间隔时间，查询两次统计值。当指定其他值时，将显示前者。

下例展示了如何使用sqlstats.sql脚本显示ID为c13sma6rkr27c的SQL语句最后15秒的统计信息：

```
SQL> @sqlstats.sql c13sma6rkr27c 15

------------------------------------------------------------------------------------------
Identification
------------------------------------------------------------------------------------------
SQL Id                                                                       c13sma6rkr27c
Execution Plan Hash Value                                                       1640444070
------------------------------------------------------------------------------------------
Shared Cursors Statistics
------------------------------------------------------------------------------------------
Total Parses                                                                             0
Loads / Hard Parses                                                                      0
Invalidations                                                                            0
Cursor Size / Shared (bytes)                                                             0
------------------------------------------------------------------------------------------
```

```
Activity by Time
-----------------------------------------------------------------------------------------
Elapsed Time (seconds)                                                             33.559
CPU Time (seconds)                                                                  0.568
Wait Time (seconds)                                                                32.991
-----------------------------------------------------------------------------------------
Activity by Waits
-----------------------------------------------------------------------------------------
Application Waits (%)                                                               0.000
Concurrency Waits (%)                                                               0.000
Cluster Waits (%)                                                                   0.000
User I/O Waits (%)                                                                 97.994
Remaining Waits (%)                                                                 0.313
CPU (%)                                                                             1.692
-----------------------------------------------------------------------------------------
Elapsed Time Breakdown
-----------------------------------------------------------------------------------------
SQL Time (seconds)                                                                 33.559
PL/SQL Time (seconds)                                                               0.000
Java Time (seconds)                                                                 0.000
-----------------------------------------------------------------------------------------
Execution Statistics                           Total          Per Execution       Per Row
-----------------------------------------------------------------------------------------
Elapsed Time (milliseconds)                   33,559                     23         5.133
CPU Time (milliseconds)                          568                      0         0.087
Executions                                     1,436                      1         0.220
Buffer Gets                                   43,305                     30         6.624
Disk Reads                                     4,292                      3         0.656
Direct Writes                                      0                      0         0.000
Rows                                           6,538                      5         1.000
Fetches                                        1,440                      1         0.220
Average Fetch Size                                 5
-----------------------------------------------------------------------------------------
Other Statistics
-----------------------------------------------------------------------------------------
Executions that Fetched All Rows (%)                                                  100
Serializable Aborts                                                                     0
-----------------------------------------------------------------------------------------
```

4.5 小结

本章介绍了一个发生性能问题时用于定位性能问题的分析路线图，同时还介绍了几种可以使用的工具和技术。尽管提供的分析路线图很有帮助，但它也只是冰山一角。总之，找到一种妥善的处理方法来快速成功地定位问题才是最重要的。对此，我已经强调过很多次了。

本章介绍了发生性能问题时该如何分析。但如果问题发生在过去呢？你能找出发生了什么，并且防止其再次发生吗？第5章将介绍如何使用包含历史性能统计信息的知识库为这些问题找到答案。

第5章

不可重现问题的事后分析

本章将介绍如何分析一个无法重现或监控到的性能问题。换句话说，当问题发生过后，无法使用SQL跟踪，也无法查看动态性能视图，这种情况下该如何分析问题。在这种情况下，能够在你想要分析的时间段做出可靠分析的唯一方法就是使用包含性能统计信息的知识库。

5.1 知识库

Oracle数据库提供了两个知识库，其中存储的信息可以用于分析过去发生过的性能问题：

- Automatic Workload Repository（AWR）
- Statspack

由于AWR是Statspack的进化版，因此它也基于以下三个同样的基本概念（这些概念就是随其提供的实用程序）。

- 在固定的间隔里（例如30分钟），许多动态性能视图的内容被导入一组表中。产生的结果数据被称为快照（snapshot），快照通过快照 ID来进行识别。有些动态性能视图会导出所有数据，有些则只会导出一部分数据。例如，SQL语句的信息只会导出消耗最大的。
- 针对AWR，可以通过Oracle提供的脚本或者工具（例如，Enterprise Manager或SQL Developer）找出在两个快照限定的时间段内，知识库中统计信息的变化情况。
- 通常情况下，快照不会无限期地保存下去，经过一段时间后就会被删除。指定时间段的快照可以标记成基线（baseline），这样就不会被删除。基线可以用来做对比。例如，如果你在系统运行良好的时候保存了一段时间的基线，就可以在性能问题发生时与基线的时间段做对比。

注意，选取快照的间隔长度是非常重要的。实际上，通常更短的间隔要比一小时甚至更长的间隔有用。这主要有两个原因。首先，对一个很长的时间段计算比率或平均值会造成很大的误导。其次，鉴于一些动态性能视图提供的信息变化非常快，在获取快照的时候，有用的信息或许已经不在了。例如，一条消耗了大量资源的SQL语句，可能会在快照捕获前从库缓存中移除，从而导致快照没有记录到它。因此，我通常建议时间间隔为20或30分钟。

表5-1总结了AWR与Statspack之间的主要区别。鉴于AWR要比Statspack强大的多，在有许可的情况下你更应该使用AWR。

表5-1 AWR与Statspack之间的主要区别

Automatic Workload Repository	Statspack
与数据库紧密整合，并且自动安装和管理	需要DBA手动安装和管理
基于ASH保存系统级别、SQL语句级别以及会话级别的信息	只保存系统和SQL语句级别的信息
Enterprise manager可以管理其内容	Enterprise Manager没有集成
自动诊断性能问题会参考其内容	不会参考其内容
需要Oracle诊断包组件和企业版	所有版本都可使用
不能在只读模式的备用数据库上使用	11.1之后的版本可以在只读模式的备用数据库上使用

5.2 自动工作负载存储库

本节将介绍如何配置AWR，捕获快照并管理基线。稍后的5.4节将会介绍如何利用存储在AWR中的信息。此时重要的是要知道AWR中存储的信息是通过dba_hist前缀的数据字典视图（在12.1多租户环境下，也存在cdb_hist前缀的视图）公开的。

5.2.1 执行配置

AWR会在每个数据库上自动安装并配置。因此从它存在之初，数据库引擎就会捕获记录工作负荷的快照。

警告 当初始化参数statistics_level设置成basic时，数据库引擎不会自动捕获快照。

配置基于以下三个参数。

- Snapshot interval：两个快照之间的时间间隔（单位：分钟）。最小值和最大值分别是10分钟和100年。默认是1小时。
- Retention period：快照的保存时间（单位：分钟）。最小值和最大值分别是1天和100年。如果指定0，那么快照会永久保存。在10.2版本中默认值是7天，11.1版本之后默认值是8天。
- Top SQL statements：每个快照都会记录消耗最大的SQL语句数量。鉴于每个快照会记录多个消耗种类（例如，top elapsed time、top CPU utilization和top parse calls），因此每个快照实际保存的SQL语句数量要比参数指定的值高。该参数的默认值（DEFAULT）是30，最大值（MAXIMUM）是50 000。这里DEFAULT可以是30或100，这要根据捕获快照的flush level来定（参见5.2.2节）。

彩色SQL ID

为了保证指定的SQL语句信息在每个快照里都捕获到（无论它是否为消耗最大的SQL），从11.1版本之后，可以将语句的SQL ID标记为colored。可以使用dbms_workload_repository包下的add_colored_sql和remove_colored_sql过程分别标记或者取消标记SQL ID的colored。请注意两个过程都需要指定操作的SQL ID。

要知道哪些SQL语句被标记为colored，可以查询dba_hist_colored_sql视图，在12.1多租户环境下，可以查询cdb_hist_colored_sql视图。

下面的查询展示了如何显示参数的当前值（注意从11.1版本之后这些值是默认值）：

```
SQL> SELECT snap_interval, retention, topnsql
  2  FROM dba_hist_wr_control;

SNAP_INTERVAL      RETENTION          TOPNSQL
----------------- ----------------- -------
+00000 01:00:00.0 +00008 00:00:00.0 DEFAULT
```

可以使用dbms_workload_repository包下的modify_snapshot_settings过程来修改默认配置。例如，以下调用设置间隔时间为20分钟，保存35天：

```
dbms_workload_repository.modify_snapshot_settings(
  interval  => 20,
  retention => 35*60*24,
  topnsql   => 'DEFAULT'
);
```

AWR的数据会保存在sysaux表空间中。存储空间的大小完全取决于这三个参数如何设置。通常情况下，每个快照会至少占用1兆空间。如果你想知道当前使用了多少空间，可以执行以下查询：

```
SELECT space_usage_kbytes
FROM v$sysaux_occupants
WHERE occupant_name = 'SM/AWR'
```

5.2.2 捕获快照

快照除了可以由数据库引擎自动捕获外，也可以手工捕获。想要保存特定时间段的信息时，快照会很有帮助。要捕获快照，需要调用dbms_workload_repository包下的create_snapshot子程序。这个包下有两个子程序：一个函数和一个存储过程。它们都需要一个参数用来指定flush level（TYPICAL或ALL，前者是默认值）。如果使用TYPICAL，会保存每个分类的前30个top SQL语句。如果指定ALL，会保存前100个。函数和存储过程唯一的不同是函数会返回快照ID。以下查询显示指定flush level为ALL的情况下如何捕获快照并显示关联的快照ID：

```
SQL> SELECT dbms_workload_repository.create_snapshot(flush_level => 'ALL') AS snap_id
  2  FROM dual;

   SNAP_ID
----------
       738
```

保存在AWR中的快照可以通过dba_hist_snapshot视图查看，在12.1多租户环境下，可以查看cdb_hist_snapshot视图：

```
SQL> SELECT begin_interval_time, end_interval_time,
  2         decode(snap_level, 1, 'TYPICAL', 2, 'ALL', snap_level) AS snap_level
  3  FROM dba_hist_snapshot
  4  WHERE snap_id = 738;

BEGIN_INTERVAL_TIME       END_INTERVAL_TIME         SNAP_LEVEL
------------------------- ------------------------- ----------
22-APR-14 04.00.22.234 PM 22-APR-14 04.06.58.230 PM ALL
```

5.2.3 管理基线

基线是由多个连续的快照组成的。基线有两种。

- 固定基线（fixed baseline）：由静态的开始快照ID和静态的结束快照ID限定的一组连续快照。可以根据需要创建多个基线。
- 移动窗口基线（moving window baseline）：指定时间内（特别是以天为单位）的一组连续快照并且以最近的快照作为结束。每个数据库都有一个移动窗口基线作为数据引擎的自适应阈值（更多信息请参考*Performance Tuning Guide*手册）。这种基线是从11.1版本之后才有的。

1. 管理固定基线

dbms_workload_repository包提供了多个命名为create_baseline的函数和存储过程用来创建基线。虽然它们在两个方面有所区别，但都实现同样的基本功能。首先，基线的起始和结束可以指定两个ID或两个时间（后者只有在11.1版本之后才可用）。其次，函数会返回新创建的基线ID。注意所有的子程序都需要参数指定基线的名称，同时也可使用可选参数指定基线在多少天后自动删除（默认值为NULL，指定没有过期的基线）。例如，以下调用展示了如何创建名为TEST的基线并指定基线于30天后过期：

```
dbms_workload_repository.create_baseline(
  start_snap_id => 738,
  end_snap_id   => 739,
  baseline_name => 'TEST',
  expiration    => 30
);
```

保存在AWR中的快照可以通过dba_hist_baseline视图查看，在12.1多租户环境下，可以通过cdb_hist_baseline视图查看：

```
SQL> SELECT start_snap_id, start_snap_time, end_snap_id, end_snap_time
  2  FROM dba_hist_baseline
  3  WHERE baseline_name = 'TEST'
  4  AND baseline_type = 'STATIC';

START_SNAP_ID START_SNAP_TIME          END_SNAP_ID END_SNAP_TIME            EXPIRATION
------------- ------------------------ ----------- ------------------------ ----------
          738 22-APR-14 04.06.58.230 PM        739 22-APR-14 04.12.50.933 PM         30
```

dbms_workload_repository包提供了select_baseline_metric函数用来显示基线（同样也对移动窗口基线适用）相关的度量值。以下查询展示了如何显示TEST基线相关的度量值：

```
SQL> SELECT metric_name, metric_unit, minimum, average, maximum
  2  FROM table(dbms_workload_repository.select_baseline_metric('TEST'))
  3  ORDER BY metric_name;

METRIC_NAME             METRIC_UNIT             MINIMUM    AVERAGE    MAXIMUM
----------------------- ---------------------- ------- ---------- ----------
Active Parallel Sessions Sessions                     0          0          0
Active Serial Sessions  Sessions                      0 1.42857143          7
Average Active Sessions Active Sessions               0 .413742101 3.49426268
...
```

```
...
User Transaction Per Sec Transactions Per Second       0 6.98898086 49.2425504
VM in bytes Per Sec      bytes per sec                 0          0          0
VM out bytes Per Sec     bytes per sec                 0          0          0
```

dbms_workload_repository包提供了rename_baseline过程来对基线重命名。该过程需要参数指定旧命名与新命名。例如，以下调用显示如何将TEST基线改名为TEST1：

```
dbms_workload_repository.rename_baseline(
  old_baseline_name => 'TEST',
  new_baseline_name => 'TEST1'
);
```

最后，可以使用dbms_workload_repository包下的drop_baseline过程来删除基线。该过程需要指定参数基线名，以及指定是否删除与基线相关的快照（默认情况下不删除）的可选参数。例如，以下调用展示了如何删除TEST1基线及其相关的快照：

```
dbms_workload_repository.drop_baseline(
  baseline_name => 'TEST1',
  cascade       => TRUE
);
```

2. 管理移动窗口基线

移动窗口基线没有太多需要管理的内容。实际上，我们仅可以调用dbms_workload_repository包下的modify_baseline_window_size过程来修改窗口大小。参数需要指定新窗口大小的天数。比如，以下调用展示了如何将窗口大小设置为30天：

```
dbms_workload_repository.modify_baseline_window_size(window_size => 30);
```

调用modify_baseline_window_size过程唯一需要满足的要求是，新的窗口大小不能大于使用快照的保存期。如果没有满足要求，数据库引擎会抛出如下异常：ORA-13541: system moving window baseline size greater than retention。

可以使用以下查询来显示当前窗口大小（注意，从11.1版本开始这些值为默认值）：

```
SQL> SELECT baseline_name, moving_window_size
  2  FROM dba_hist_baseline
  3  WHERE baseline_type = 'MOVING_WINDOW';

BASELINE_NAME        MOVING_WINDOW_SIZE
-------------------- ------------------
SYSTEM_MOVING_WINDOW                  8
```

5.3 Statspack

本节介绍如何安装和配置Statspack、捕获快照和管理基线。稍后在5.5节中将介绍如何利用存储在Statspack知识库中的信息。

提示 Oracle数据库手册不再提供关于Statspack的信息。可以在$ORACLE_HOME/rdbms/admin目录下的spdoc.txt文件中找到安装、配置、管理和使用Statspack（以及其他安装脚本和工具）的详细信息。Oracle Support文档*Installing and Using Standby Statspack in 11g*（454848.1）提供了关于在只读模式的备用数据库上使用Statspack的信息。

5.3.1 执行安装

为了安装Statspack，需要以sys用户身份连接数据库，并运行$ORACLE_HOME/rdbms/admin目录下的spcreate.sql脚本。该脚本会创建perfstat用户以及用户下大部分需要运行Statspack的对象。此外，它还会创建多个public同义词和sys模式下的视图。执行期间，脚本会询问perfstat用户的密码，使用的临时表空间和存放表和索引的表空间。注意，在执行脚本前，要指定的临时表空间和表空间就需要创建好。如果不希望创建新的表空间，可以选择默认的临时表空间和sysaux表空间。

5

5.3.2 配置存储库

Statspack的配置基于三类参数，保存在perfstat模式的stas$statspack_parameter表下。

- 快照级别（Snapshot level）：定义捕获快照时存储的数据。表5-2简单介绍了可用的快照级别。同样，在stats$level_description表里也包含对每个级别的简介。

表5-2 Statspack快照级别

级 别	描 述
0	捕获一般性能统计信息
5	捕获一般性能统计信息（同级别0），也包括超过阈值的SQL语句统计信息。这是默认级别
6	除了低级别收集的所有统计信息外，还包括执行计划（包括使用统计信息）
7	除了低级别收集的所有统计信息外，还包括超过阈值的段级别统计信息（比如，逻辑读和物理读的数量）
10	除了低级别收集的所有统计信息外，还包括闩的统计信息

- SQL语句阈值（SQL statement threshold）：有六个阈值（执行数、解析调用数、物理读数、逻辑读数、可共享内存数和子游标数）用来判断是否捕获SQL语句。只有至少超过其中一个阈值时才会捕获SQL语句。
- 段统计信息阈值（Segment statistics threshold）：有七个阈值（逻辑读数、物理读数、buffer busy wait数、row lock wait数、ITL wait数、全局缓存一致性读块数和全局缓存当前块数）用来判断是否捕获段信息。只有至少超过其中一个阈值时才会捕获段信息。

以下查询展示了每个参数的实际值（这里显示的是安装过后的默认值）：

```
SQL> SELECT parameter, value
  2  FROM stats$statspack_parameter
  3  UNPIVOT (
  4    value FOR
  5    parameter IN (snap_level, executions_th, parse_calls_th, disk_reads_th, buffer_gets_th,
  6                  sharable_mem_th, version_count_th, seg_phy_reads_th, seg_log_reads_th,
```

```
  7                  seg_buff_busy_th, seg_rowlock_w_th, seg_itl_waits_th, seg_cr_bks_rc_th,
  8                  seg_cu_bks_rc_th)
  9  );

PARAMETER          VALUE
---------------- -------
SNAP_LEVEL             5
EXECUTIONS_TH        100
PARSE_CALLS_TH      1000
DISK_READS_TH       1000
BUFFER_GETS_TH     10000
SHARABLE_MEM_TH  1048576
VERSION_COUNT_TH      20
SEG_PHY_READS_TH    1000
SEG_LOG_READS_TH   10000
SEG_BUFF_BUSY_TH     100
SEG_ROWLOCK_W_TH     100
SEG_ITL_WAITS_TH     100
SEG_CR_BKS_RC_TH    1000
SEG_CU_BKS_RC_TH    1000
```

statspack包提供了modify_statspack_parameter过程来修改参数的值。针对每个参数，存储过程都有一个输入参数。比如，以下调用将快照级别更改为6：

```
statspack.modify_statspack_parameter(i_snap_level => 6)
```

5.3.3　捕获和清除快照

你可以调用statspack包下的snap子程序来捕获快照。包下有两个子程序：一个函数和一个存储过程。它们都可以指定前面部分介绍的任意一个配置参数。所有的参数都是可选的，因此可以不指定任何参数而捕获快照，比如下面这样：

```
perfstat.statspack.snap();
```

只要不是被定义成基线的快照，都可以使用statspack包下的purge子程序来清除。包下一共有八个子程序。一方面，函数和过程执行着同样的功能。另一方面，有四种方法可用来指定清除快照：

- 指定起始和结束快照ID之间的所有快照（参数i_begin_snap和i_end_snap）；
- 指定开始和结束时间之间的所有快照（参数i_begin_date和i_end_date）；
- 指定某一时间之前的所有快照（参数i_purge_before_date）；
- 指定超过特定天数的所有快照（参数i_num_days）。

注意，默认情况下不会清除SQL语句和执行计划的数据。如果要清除这些数据，需要设置参数i_extended_purge为TRUE来激活extended purge。比如，以下调用会清除2014年4月之前的所有快照（包括SQL语句和执行计划）：

```
statspack.purge(
  i_purge_before_date => to_date('2014-04-01','YYYY-MM-DD'),
  i_extended_purge    => TRUE
);
```

鉴于快照不能自动捕获也无法在特定时间后清除，所以需要计划两个任务来执行这些操作。请看

下面的例子（注意，两个任务都应由perfstat用户来创建）。

- 每隔15分钟捕获一次快照。

```
dbms_scheduler.create_job(
  job_name        => 'TAKE_STATSPACK_SNAPSHOT',
  job_type        => 'PLSQL_BLOCK',
  job_action      => 'perfstat.statspack.snap();',
  start_date      => sysdate,
  repeat_interval => 'FREQ = HOURLY; BYMINUTE = 0,15,30,45',
  enabled         => TRUE,
  comments        => 'take STATSPACK shapshot'
);
```

- 清除创建超过35天的快照。

```
dbms_scheduler.create_job(
  job_name        => 'PURGE_STATSPACK_SNAPSHOTS',
  job_type        => 'PLSQL_BLOCK',
  job_action      => 'statspack.purge(i_num_days => 35, i_extended_purge => TRUE);',
  start_date      => sysdate,
  repeat_interval => 'FREQ = HOURLY; BYMINUTE = 50',
  enabled         => TRUE,
  comments        => 'purge STATSPACK shapshots'
);
```

可以在spauto.sql和sppurge.sql脚本中找到Oracle提供的其他例子。要获取这两个脚本，可查找$ORACLE_HOME/rdbms/admin目录。

注意 在RAC环境下，需要在每个数据库实例上分别计划任务。

5.3.4 管理基线

为常规快照设置标记，标记上baseline的快照是基线。因此，它不同于删除处理。statspack包提供了两组存储过程和函数来管理这些标记。第一组由make_baseline的子程序组成，用来为一个或多个快照设置标记。第二组由clear_baseline的子程序组成，用来为一个或多个快照取消标记。statspack包提供了同样功能的函数和存储过程。唯一的区别是，函数会返回标记修改的快照数量。可以指定一系列ID或一段时间来确定需要修改的快照。例如，以下调用会标记两个指定时间戳之间的快照为基线：

```
perfstat.statspack.make_baseline(
  i_begin_date => to_date('2014-04-02 17:00:00','YYYY-MM-DD HH24:MI:SS'),
  i_end_date   => to_date('2014-04-02 17:59:59','YYYY-MM-DD HH24:MI:SS')
);
```

基于同样的方法，以下调用会取消两个指定时间戳之间的所有快照标记：

```
perfstat.statspack.clear_baseline(
  i_begin_date => to_date('2014-04-02 00:00:00','YYYY-MM-DD HH24:MI:SS'),
  i_end_date   => to_date('2014-04-02 23:59:59','YYYY-MM-DD HH24:MI:SS')
);
```

5.4 使用 Diagnostics Pack 进行分析

要使用Diagnostics Pack进行分析，建议使用Enterprise Manager的performance页面（这与你使用的是Database Control、Grid Control还是Cloud Control无关）。正如第4章描述的那样，Performance Home和Top Activity页面显示实时和历史信息。鉴于本章旨在分析发生过的性能问题，所以会用到历史信息。在这种模式下，默认情况下信息可以保留一周。

历史信息分析与实时数据分析基本相同。因此，详细信息请参考第4章。这里主要介绍两者的不同之处。

- 利用历史数据，显示在30分钟间隔里（不是5分钟）数据库负载的详细信息。（想要更灵活，可以使用ASH Analytics）。
- 由于不是所有实时数据都保存在AWR中，因此数据会缺少详细信息，甚至缺失详细信息。不过你可以访问到足够多的关于最高负载的信息。
- 无法搜索特定的会话。实际上，Performance菜单里的Search Sessions选项不可用。会话只能通过Top Sessions表访问。

除了Enterprise Manager集成的部分，AWR提供了一系列报告用来评估指定的时间段内的负载情况。下面是三个最常用的报告。

- AWR报告总结了一段时间内指定的操作。这个报告可以由Enterprise Manager生成，也可以执行`$ORACLE_HOME/rdbms/admin/awrrpt.sql`脚本来实现。鉴于该报告基于Statspack报告，请参考下一部分中关于它的解释信息。
- 周期对比报告对比两个指定的独立时间段。这对找出数据库引擎经历性能问题的时间段与基线时间段之间的区别非常有用。这个报告可以由Enterprise Manager生成，也可以通过执行`$ORACLE_HOME/rdbms/admin/awrddrpt.sql`脚本来实现。
- SQL语句报告提供关于SQL语句的详细信息。更多信息请参考第10章，特别是10.1.3节。

5.5 不使用 Diagnostics Pack 进行分析

如果不关注单条SQL语句，那么，借助Statspack，通过执行`$ORACLE_HOME/rdbms/admin/spreport.sql`脚本可开始对性能问题的分析。脚本在询问生成报告的时间段（我建议选择两个连续的快照）之后会把报告写入一个输出文件中。尽管这部分的目标是介绍如何阅读报告，但是要做到面面俱到也是不可行的。实际上，报告不仅非常长（一般有2000~3000行），并且可能包含许多大部分时间都不需要去关注的内容。很多内容只是为了应对不时之需。

提示　AWR是Statspack的进化版，因此对Statspack报告的解释阅读也同样适用于AWR报告。

分析从报告的最初部分开始（大约100行）。这部分信息排列得不是特别好，也不是很有趣。然而，由于报告的最初部分不是很长，却也值得去阅读。

Statspack报告首先介绍了实例和承载服务器的自述信息。

```
Database    DB Id      Instance     Inst Num Startup Time    Release    RAC
~~~~~~~~ ----------- ------------ -------- --------------- ----------- ---
          2532911053 DBM11203            1 23-Apr-14 16:33 11.2.0.3.0  NO

Host Name             Platform               CPUs Cores Sockets   Memory (G)
~~~~ ---------------- ---------------------- ----- ----- ------- ------------
     helicon          Linux x86 64-bit           8     8       2          7.8
```

对于上面的摘要，请记住数据库服务器的CPU核数。稍后，这一信息会用来评估数据库引擎是否是CPU bound。

警告 当数据库服务器配置同步多线程CPU时，v$osstat视图的NUM_CPUS的值（也是Statspack报告中的CPUs列的值）会被设成线程总数。注意使用线程数来评估数据库是否为CPU bound会造成误导。实际上，100%使用所有线程是不可能的。即使基于CPU核数的检查会被认为太过保守，却也可以放心地使用（假如你无法在虚拟化环境下这么做）。

报告接下来提供了指定时间段（开始、结束和持续时间）的信息，开始时间段和结束时间段内的会话数，在11.1版本之后还会有DB time（106.55）和CPU使用率（28.35）。在这一部分，需要仔细查看需要分析的时间段内的报告。注意：平均活动会话数（7.1）只在11.1.0.7及以后的版本中存在，它是DB time除以elapsed time的值。

```
Snapshot       Snap Id     Snap Time      Sessions Curs/Sess Comment
~~~~~~~~    ---------- ------------------ -------- --------- ------------------
Begin Snap:        548 23-Apr-14 18:30:40       57       1.6
  End Snap:        549 23-Apr-14 18:45:40       59       1.5
   Elapsed:      15.00 (mins) Av Act Sess:       7.1
   DB time:     106.55 (mins) DB CPU:           28.35 (mins)
```

报告接下来提供了最重要的SGA组件的大小。注意缓冲区缓存（buffer cache）或共享池在观测的时间段内是否发生改变，如果发生了改变，那么结束时间的值也会显示。否则，就像下面的例子这样，只会显示开始时间的值：

```
Cache Sizes            Begin        End
~~~~~~~~~~~       ---------- ----------
    Buffer Cache:       728M              Std Block Size:         8K
     Shared Pool:       260M                  Log Buffer:     7,992K
```

接下来会看到大量的度量值，包括每秒处理、每个事务处理以及每次执行和调用的度量值：

```
Load Profile              Per Second    Per Transaction    Per Exec    Per Call
~~~~~~~~~~~~      ------------------  ----------------- ----------- -----------
      DB time(s):                7.1                0.0        0.01        0.00
       DB CPU(s):                1.9                0.0        0.00        0.00
       Redo size:          392,163.4            1,928.3
   Logical reads:          406,805.1            2,000.3
   Block changes:            2,822.9               13.9
  Physical reads:              579.7                2.9
 Physical writes:              377.8                1.9
      User calls:            2,895.2               14.2
          Parses:                0.6                0.0
```

```
         Hard parses:              0.0                0.0
    W/A MB processed:              0.1                0.0
              Logons:              0.0                0.0
            Executes:          1,364.1                6.7
           Rollbacks:              0.0                0.0
        Transactions:            203.4
```

上面大多数的度量值无法直接用于评估数据库实例是否存在性能问题。它们的主要目的有两个。第一，让我们对负载有个大体了解。例如，在上面的例子中，我们看到每秒事务量为203.4，并且平均每个事务发生了6.7次执行。因此可以知道系统正在执行某些操作。第二，可以用来判断系统做的工作是否超过预期，并且更重要的是，可以用这些值与基线进行对比。不过，有两个度量值可以用来直接判断数据库实例负载（不过，这两个值在10.2版本中不存在）。

- DB time Per Second（7.1）与平均活动会话数相等。根据第4章介绍的经验法则，若平均活动会话数与CPU内核个数（8）相等，可以认为系统相当繁忙。
- DB CPU per second（1.9）告诉你实例是否是CPU bound。实际上，可以将它与CPU内核个数（8）作对比，进而判断CPU的平均使用率。本例数据库实例仅仅消耗了24%（1.9/8 × 100）的CPU。因此，如果服务器上没有其他数据库实例或应用，那么CPU完全可以满足负载。

报告接下来是一组有限的使用率。唯一明智的做法就是用其与基线对比来判断是否有值改变：

```
Instance Efficiency Indicators
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
              Buffer Nowait %: 100.00       Redo NoWait %:  99.98
              Buffer  Hit   %:  99.86 Optimal W/A Exec %: 100.00
              Library Hit   %: 100.01        Soft Parse %:  98.99
           Execute to Parse %:  99.96         Latch Hit %:  99.96
Parse CPU to Parse   Elapsd %: 103.03     % Non-Parse CPU:  98.84

 Shared Pool Statistics        Begin   End
                               ------  ------
             Memory Usage %:   66.47   66.55
    % SQL with executions>1:   71.80   71.85
  % Memory for SQL w/exec>1:   72.70   72.88
```

接下来的部分显示top 5事件的资源使用分析（包括CPU使用率）。简单来说，在这张表里，DB time被拆分以显示时间是如何花费的。例如，根据以下的摘录，71.6%的时间花在了单块读上：

```
Top 5 Timed Events                                                Avg %Total
~~~~~~~~~~~~~~~~~~                                               wait   Call
Event                                        Waits    Time (s)   (ms)   Time
----------------------------------------- ------------ ----------- ------ ------
db file sequential read                        520,240       4,567      9   71.6
CPU time                                                     1,620          25.4
log file sync                                  182,275          94      1    1.5
log file parallel write                        178,406          39      0     .6
read by other session                            2,693          27     10     .4
```

与AWR报告相反，top 5事件的列表不会显示等待级别（例如，User I/O、System I/O、Commit或Concurrency）。如果看到一个事件属于选择忽略的等待级别，可以执行以下查询来找到它：

```
SQL> SELECT wait_class
  2  FROM v$event_name
```

```
  3  WHERE name = 'db file parallel read';

WAIT_CLASS
----------
User I/O
```

报告接下来给出操作系统级别的CPU使用率信息（注意，直到11.1.0.6版本，`Instance CPU`部分只提供百分比值并且使用不同的标签）：

```
Host CPU (CPUs: 8 Cores: 8 Sockets: 2)
~~~~~~~~         Load Average
               Begin       End     User  System    Idle     WIO     WCPU
             ------- -------  ------- ------- ------- ------- --------
                5.98    7.09    23.83    0.89   74.75   21.86
Instance CPU
~~~~~~~~~~~~
                                              % Time (seconds)
                                            -------- --------------
                        Host: Total time (s):                7,001.7
                     Host: Busy CPU time (s):                1,768.2
                      % of time Host is Busy:     25.3
                 Instance: Total CPU time (s):               1,739.9
              % of Busy CPU used for Instance:     98.4
            Instance: Total Database time (s):               6,500.3
%DB time waiting for CPU (Resource Mgr):            0.0
```

上面的摘录有以下两个目的。

- 确认主机（不是数据库实例）是否是CPU bound。`% of time Host is Busy`值提供了你需要的信息。如果这个值低（比如本例），那就没问题。如果该值高（接近100%，前提是CPU没有使用同步多线程），那么top 5事件的资源使用分析和其他等待事件的统计信息会造成误导。实际上，在CPU不足的案例中，许多统计值会被人为提高。因此，首要目标是找出方法来降低CPU使用率。
- 确认操作系统级别的CPU使用率是否主要源于所分析的数据库实例。当服务器上运行多个数据库实例或其他应用时，这是一个重要信息。最需要检查的值是`% of Busy CPU used for Instance`，该值显示总CPU使用率中有多少是由于你正在查找的数据库实例所导致的。如果该值低于80%~90%，表示有其他应用占用了过多的CPU。例如，在之前的摘要中，数据库实例几乎使用了所有的CPU（98.4%）。这代表这台服务器上没有其他应用在占用过多的CPU。

操作系统级别的CPU使用率信息之后，是操作系统级别的内存使用率信息。通过它可以知道在主机上有多少内存可用（8 GB）以及数据库实例分配给SGA和PGA的百分比（15.1%）。

```
Memory Statistics                       Begin          End
~~~~~~~~~~~~~~~~~                ------------ ------------
                  Host Mem (MB):      7,974.6      7,974.6
                   SGA use (MB):      1,019.4      1,019.4
                   PGA use (MB):        175.1        183.1
    % Host Mem used for SGA+PGA:         15.0         15.1
```

前100行最后提供的是时间模型统计信息。正如第4章讨论过的那样，基于这些数据，可以知道数据库引擎处理关键操作花费的总时间。在接下来的例子中，统计信息显示大部分时间（95.5%）是用来执行SQL语句。

```
Statistic                                 Time (s) % DB time
----------------------------------- ------------------ ---------
sql execute elapsed time                     6,103.6      95.5
DB CPU                                       1,701.1      26.6
parse time elapsed                              26.8        .4
sequence load elapsed time                       0.1        .0
PL/SQL execution elapsed time                    0.0        .0
repeated bind elapsed time                       0.0        .0
DB time                                      6,392.9
background elapsed time                        107.5
background cpu time                             38.8
```

基于报告前100行提供的信息，应该可以对当前数据库有一个相对清晰的了解，比如，系统加载的范围和执行的主要操作。下一步是找出top SQL语句。出于此目的，报告会包含多个按不同条件（CPU、elapsed time、逻辑读数、物理读数、执行数和解析调用数）排序的列表。鉴于在大多数情况下，elapsed time是最重要的指标，所以建议根据`SQL ordered by Elapsed time`部分继续分析。

在查看列表之前，需要确认捕捉到的SQL语句是否占用DB time。如果像下面引用的部分一样，捕捉到的SQL语句占用了大部分的DB time（95.4%），那么这个列表就包含有用信息：

```
-> Total DB Time (s):          6,393
-> Captured SQL accounts for  95.4% of Total DB Time
-> SQL reported below exceeded 1.0% of Total DB Time
```

然而，万一捕捉到的SQL语句只占了DB time的很小百分比（比如，10%~20%），那么这个列表用处就不大了。实际上，要么占用大量DB time的SQL语句在快照捕获前被清出了库缓存，要么就是不存在占用大量DB time的SQL语句。如果是前者，那么知识库没有包含足够的信息来完成分析。如果是后者，那么没有单独哪条SQL语句占用了过高的DB time。因此，正如4.1节所述，关注top SQL语句是没有意义的。

如接下来的引用所示，对于每条SQL语句，不仅可以看到总计和平均的elapsed time，也可以看到CPU使用率和物理读：

```
  Elapsed                    Elap per            CPU                        Old
  Time (s)    Executions    Exec (s) %Total   Time (s) Physical Reads Hash Value
---------- ------------ ---------- ------ ---------- --------------- ----------
   1861.91      124,585       0.01   29.1      32.06         194,485 3739063178
Module: Swingbench User Thread
select customer_id, cust_first_name ,cust_last_name ,nls_languag
e ,nls_territory ,credit_limit ,cust_email ,account_mgr_id  from
 customers where customer_id = :1
   1354.48        7,087       0.19   21.2    1241.76           7,834 1481390170
Module: Swingbench User Thread
SELECT /*+ first_rows index(customers, customers_pk) index(orde
rs, order_status_ix) */ o.order_id, line_item_id, product_id, u
nit_price, quantity, order_mode, order_status, order_total, sale
s_rep_id, promotion_id, c.customer_id, cust_first_name, cust_las

    648.93       36,705       0.02   10.2      20.58          70,411 3476971243
Module: Swingbench User Thread
insert into orders(ORDER_ID, ORDER_DATE, CUSTOMER_ID, WAREHOUSE_
ID) values (:1 , :2 , :3 , :4 )
```

如果正如上面的引用那样，一小部分SQL语句占用了大量的DB time（前三个SQL语句占用了超过60%的DB time），那么就需要找到这些SQL语句。然后基于散列值，使用`sprepsql.sql`脚本来获取SQL语句的所有可用信息（请参考第10章，特别是10.1.3节）。

报告的其他部分可以用来获取特定的行为或系统配置的更多详细信息。例如，在这部分介绍的一个场景中，系统是disk I/O bound，针对每个磁盘I/O操作相关的top event，应该检查`Wait Event Histogram`部分的直方图。基于直方图和I/O子系统的配置，就能够判断磁盘I/O操作是否与预期一致。根据以下引用，可以发现日志写入总少于1毫秒，而读取速度却慢得不止一个数量级。

```
                           Total ----------------- % of Waits ------------------
Event                      Waits  <1ms  <2ms  <4ms  <8ms <16ms <32ms  <=1s   >1s
-------------------------- ----- ----- ----- ----- ----- ----- ----- ----- -----
db file sequential read     520K   2.1   2.9  19.8  49.1  18.2   4.4   3.4
log file parallel write     178K  98.7    .3    .3    .2    .3    .2    .0
```


5.6 小结

本章介绍了Oracle数据库为了分析发生过的性能问题所提供的两种知识库AWR和Statspack，以及其安装、配置和管理。此外，还概括介绍了如何阅读Statspack报告（与AWR报告十分相似）。这些是本章你应该掌握的主要内容。

通常情况下，我们的目的不是调查性能问题，而是在第一时间避免它。根据我的经验，性能问题主要有两种起因：设计数据或应用时没有考虑性能问题，以及糟糕的查询优化器配置。而后者尤为关键，因为数据库引擎执行的每条SQL语句都会经过查询优化器。因此，第三部分不仅会解释查询优化器的工作原理，还会介绍如何正确配置查询优化器。

Part 3 第三部分

查询优化器

尽人事，听天命。

——爱比克泰德[①]（Epictetus）

发送到数据库的每个 SQL 语句在由 SQL 引擎处理之前都要转化成执行计划。事实上，应用程序只是通过 SQL 语句指定了什么样的数据必须处理，而未指定如何处理。查询优化器的目标不仅是提供执行计划来描述如何处理数据，同时最重要的是，交付高效的执行计划。如果做不到这一点可能会导致糟糕的性能。也正因如此，有关数据库性能的书必须涉及查询优化器。

但是，这部分的目标并不是讲述查询优化器的内部工作机制。相反，这里会呈现一个非常实际的方法，针对你必须了解的查询优化器的基本特征进行讲述。第 6 章介绍查询优化器的基本概念和体系结构。第 7、8 章讨论查询优化器使用的统计信息。第 9 章描述影响查询优化器行为的初始化参数以及如何设置它们。最后，第 10 章概述获取执行计划的不同方法，同时也介绍如何阅读它们并识别出低效的计划。

在 Oracle 数据库中，提供两个主要的查询优化器：基于规则的优化器（RBO）和基于成本的优化器（CBO）。从 Oracle Database 10*g* 开始，已经不再支持基于规则的优化器，所以我们不会涵盖这部分内容。在本书中，谈到查询优化器这个术语时，始终指的是基于成本的优化器。

① http://www.quotationspage.com/quote/2525.html。

第6章

查询优化器简介

查询优化器是SQL引擎的构成组件之一。它的用途是及时提供高效的查询计划。时间约束至关重要，因为大多数情况下在优化阶段花费过多时间都是不明智的。“过多时间”是什么意思？通常来讲，包含查询优化器执行工作的解析阶段，应该要比执行阶段时间更短。在解析阶段比执行阶段花费更长时间的众多情形中，唯一可以接受的是当一个游标可以被多次执行重用时。正如在第2章中所讨论的，在SGA中缓存与某个游标关联的共享SQL区的能力也是出于同样的目的而引入的。

本章的目的是概述查询优化器执行工作所用的信息，描述SQL引擎的体系结构，并且解释其内部组件是如何交互来处理SQL语句的。同时也提供了查询优化器中查询转换实施过程的信息。

6.1 基础知识

要选择一个执行计划，查询优化器需要回答下列问题。

- 从SQL语句引用的每张表中抽取数据的最优访问路径是什么？
- 要扫描即将处理的引用表的数据，哪一种连接方法以及连接顺序是最优的？
- 在SQL语句执行过程中，应该何时去处理聚集或排序操作？
- 使用并行处理是否有益？

然而在实践中查询优化器并不会直接回答这些问题。它会探索所谓的搜索空间来寻求最优的执行计划，搜索空间由所有潜在可行的执行计划组成。为了找出哪个执行计划是最优的，查询优化器会估算若干执行计划的成本，并从中选择成本最低的那一个。举例来说，以下查询的搜索空间包含一百多种可能的执行计划。通过search_space.sql这个脚本能够重现其中的157种。它们的成本从20一直到100 000多。

```
SELECT *
FROM t1 JOIN t2 ON t1.id = t2.t1_id
WHERE t1.n = 1 AND t2.n = 2
```

因为查询优化器的目标是尽可能迅速地找出成本最低的执行计划，除了最简单的SQL语句，查询优化器并不会评估所有的执行计划，这一点至关重要。换句话说，查询优化器只探索搜索空间的一部分。简而言之就是基于启发式的选择，查询优化器从评估最有希望的执行计划开始，然后考虑其他的执行计划直到成本最低的那个出现，或者探查到太多可供选择的执行计划。它实现了一个叫作分支定界（branch-and-bound）的算法。一个分支就是一个可供选择的执行计划（例如，一条访问路径或者一个连接方法），而边界则是到目前为止找到的最佳执行计划的成本，一旦发现当前分支的成本比边界高，查询优化器就会尽快丢弃它（并很可能连带丢弃其所有子分支）。

图6-1展示了如何估算一个执行计划的成本，查询优化器不仅要考虑所要优化的SQL语句，还要考虑其他若干输入信息。其中有些输入信息存储在数据字典中而且几乎很少改变，或者说在运行时不期望其经常改变。当应用程序运行时可以认为它们是静态的环境变量。而另外的一些输入项不仅可能会经常改变，甚至每次执行时都会改变，还可能直到运行时之前都不知道是否会改变。正因为这些输入项，对于一条给定的SQL语句，查询优化器可能在每次处理时都产生一个新的执行计划。

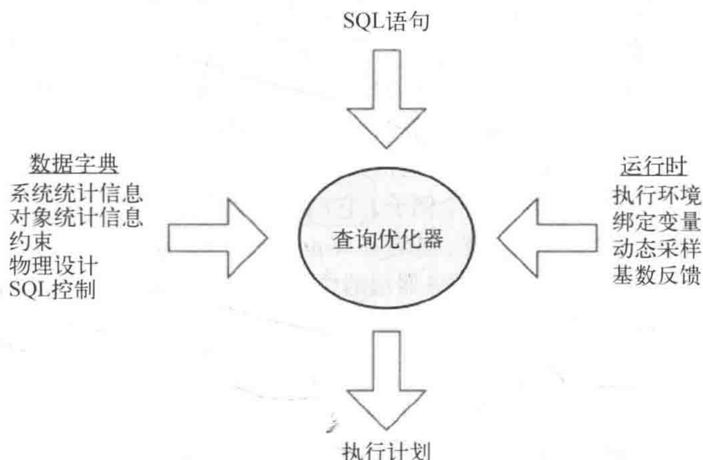


图6-1 查询优化器参考若干输入项来产生执行计划

图6-1中的一些输入项是用来判定哪些选项是可用的。其他的用来估算潜在执行计划的成本。下面的列表简要描述了这些输入项，并指出了本书的哪些部分提供关于它们的详细信息。

- **系统统计信息**：查询优化器必须知道它所运行的系统的能力才能提供精确的估算。为此，系统统计信息既描述运行数据库引擎的机器，同时也给出存储子系统的性能指标。第7章介绍了有哪些系统统计信息可用，如何管理它们，以及查询优化器如何利用它们来改进估算。
- **对象统计信息**：表、索引以及列统计信息，这些存储在数据字典中的信息很关键，因为它们描述了存储在数据库中的数据情况。例如，仅仅知道将要处理的SQL语句和引用对象的结构，查询优化器无法提供高效的执行计划。为了产生高效的执行计划，查询优化器必须能够量化所要处理的数据总量，还要知道通过各种不同的可选项对数据进行处理的成本。第8章描述了有哪些对象统计信息可以使用，以及如何管理它们。
- **约束**：查询优化器利用非空约束、唯一键约束、主键约束、外键约束以及一些检查约束。本章稍后会讲述，约束对于评估应用特定的查询转换是否有可能或者是否合理也很关键。此外，因为这些原因，在定义存储在数据库中的对象时，要创建所有已知的约束，这样做是很明智的。
- **物理设计**：有三个主要的物理设计领域对查询优化器有影响。第一，Oracle数据库提供了五种存储数据的策略：堆组织表（默认表类型）、索引组织表、外部表、索引聚簇和散列聚簇。另外，堆组织表和索引聚簇可以进行分区。每种策略都有一条或多条访问路径与之关联。第13章会详细介绍这些访问路径。第二，对于除了外部表以外的每种策略，Oracle数据库都可以处理多种类型的索引。每种索引类型（参见第13章）都会增加具体的访问路径。此外，所有的存储策略都支持物化视图，以便通过查询重写给予查询优化器额外的途径来优化查询。这个

主题会在第15章中进行讨论。第三，即使列顺序并不影响访问路径，但是会影响查询优化器的一些成本的计算。这背后的原因可以在第7章和第16章中找到答案。

- **SQL控制**：大多数情形下，查询优化器能够生成最优的执行计划。但也有查询优化器做不到的例外情况，Oracle提供了一些特性来改善这些情况出现时带来的麻烦。第11章会详细讨论这些特性。目前重要的是，要知道类似存储基线、SQL概要和SQL计划基线这样的特性允许你将它们存入数据字典信息中，从而在查询优化器产生执行计划时影响它的某些决定。
- **执行环境**：有一组初始化参数控制着查询优化器的行为。这些参数通过数据库引擎的初始值或者服务器参数文件SPFILE设置在系统级别。如果有需要，这些参数可以通过在会话级别发出ALTER SESSION语句来覆盖之前的值。第11章会讲到其中的一部分甚至可以在SQL语句级别进行更改。有些参数可以在操作系统级别将其设置在服务器端，也可以设置在客户端。国家语言支持（NLS）参数就是这样的一个例子，它们可以配置在连接的两端。实际上，NLS参数也可以通过操作系统环境变量来设置，或者在Windows上通过注册表设置。特别是在客户端设置时，你必须很小心：对于客户端/服务器端的应用，有一些客户端的环境变量会对查询优化器产生影响，这一点经常被忽略。第9章会讨论控制查询优化器行为的最重要的初始化参数。在第13章中会涉及一部分NLS参数。
- **绑定变量**：绑定变量已经在第2章进行了完整的介绍。除了值以外，绑定变量的定义（也就是数据类型）也会对查询优化器生成执行计划造成强烈影响。
- **动态采样**：根据存储在数据字典中的对象统计信息，查询优化器并不总是能够精确地估算出某个操作或者谓词的成本。当查询优化器识别出这样的案例，在某些情形下它能够在执行查询优化期间动态采集额外的统计信息。要这样做，查询优化器会针对待优化SQL语句引用的对象执行递归查询。这一特性会在第9章中进行介绍。
- **基数反馈（也称为统计信息反馈）**：不管是因为复杂的谓词还是缺少输入信息，查询优化器并不总是能够进行精确的估算。当查询优化器意识到它正在为一个SQL语句进行低质量的估算，那么生成的执行计划会带有注解。在SQL语句执行完毕后会对估算的准确性进行检查。如果实际值和估算值差异明显，正确的值的信息就会被存储，并在SQL语句下次执行时强制再优化。注意再优化会强制查询优化器利用第一次执行时获得的信息，进而增加了生成最优执行计划的机会。这个特性仅从11.2版本起可用。

使用的Oracle数据库软件的版本也会决定哪些查询优化器的特性可用。要清楚认识到，即便知道了Oracle数据库版本的五位数字，对于完全了解哪些特性可用也是不够的。还需要知道正在使用的版本是企业版还是标准版。此外，注意，某些补丁也会控制特定查询优化器特性的可用性以及其的行为。

6.2 体系结构

查询优化器可以分解为逻辑优化器和物理优化器，它只是SQL引擎的一个构成模块。

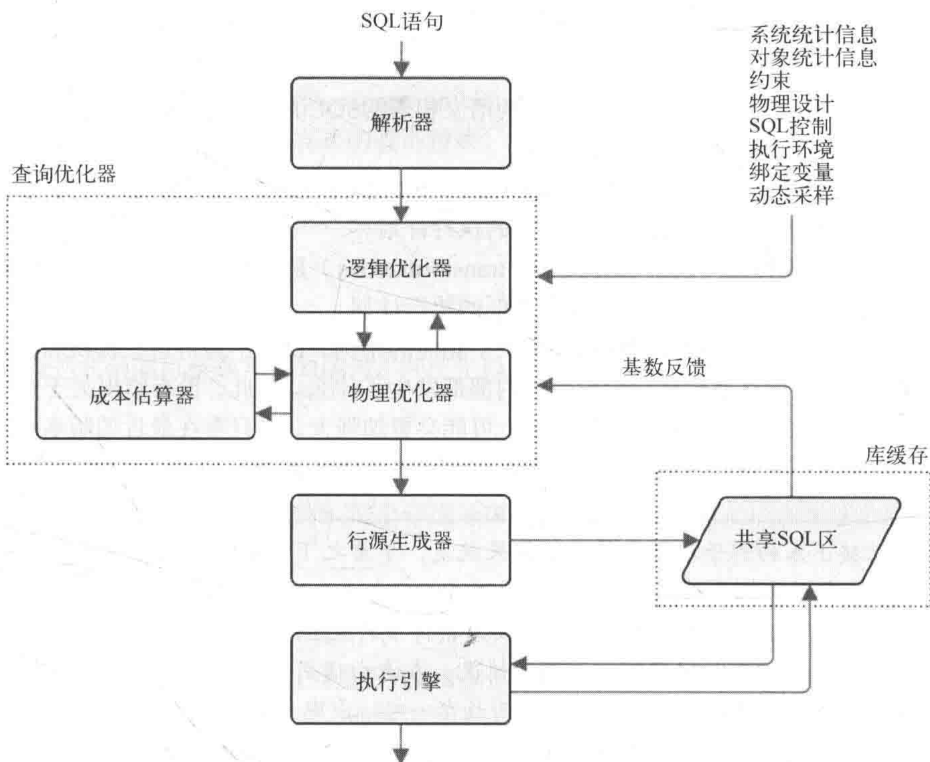


图6-2 SQL引擎的体系结构

如图6-2所示，以下是SQL引擎的关键组件。

- **解析器**：这是与SQL执行有关的第一个组件。它的用途是向查询优化器传递SQL语句解析后的形式。关于解析器执行工作的更多信息已经在第2章尤其是2.4节中介绍过。
- **逻辑优化器**：在逻辑优化阶段，查询优化器通过应用不同的查询转换技术产生新的语义相等的SQL语句。逻辑优化器的目的是选择出查询转换的最佳组合。在这种情况下，搜索空间增加了，执行计划可以被探索而不会被认为没有经过这样的查询转换。6.3节会提供关于这个组件执行工作的额外信息。
- **物理优化器**：在物理优化阶段，执行了几项操作。一开始，针对由逻辑优化生成的每个SQL语句生成了几个执行计划。然后每一个执行计划都发送给成本估算器让其计算出一个成本。最后，拥有最低成本的那个执行计划就被选中了。简单地说，物理优化器探索搜索空间来找出最有效率的执行计划。
- **成本估算器**：根据图6-1中介绍的输入项，成本估算器计算由物理优化器提交的执行计划的成本。
- **行源生成器**：查询优化器生成的执行计划不能直接由执行引擎执行。它必须转化成行源操作树以存储在库缓存中。
- **执行引擎**：这个组件执行由行源生成器产生的行源操作。如果基数反馈的监控是激活的，执行引擎（执行完毕后）会校验实际值和估算值的差异是否明显。如果找到了明显差异，关于正确值的信息就会存储到共享SQL区中，并且在下一次执行中，再优化是强制进行的。

6.3 查询转换

查询优化器使用大量的查询转换来产生新的语义相等的SQL语句。在那些查询转换中，根据用于决定是否应用它们的方法，可以分为两种途径。

- **基于启发式的查询转换**（Heuristic-based query transformations）是在满足特定条件时应用的。在大多数情况下它们预计都会引出更好的执行计划。
- **基于成本的查询转换**（Cost-based query transformations）是根据成本估算器计算的成本而应用的，它们会引出与原始语句相比成本更低的执行计划。

下面的章节介绍了二十几种查询转换并提供了相应的用例。目的不是对这些转换高谈阔论，而是为你提供一些头绪去了解在逻辑优化阶段引擎内部都发生了什么。因此，没有给出关于先决条件或限制性的详细信息。同时注意一些查询转换实际上可能会更加强大，或只有在最近的版本中才可用。所以，并非本章描述的每种查询转换在早前的版本中都可用，比如在版本10.2和11.1中。

注意　我试图让接下来的例子尽可能简单。结果就是，乍看之下，某些查询转换似乎只有在处理劣质的SQL语句时才可能用得到。劣质的意思是指包含多余或冲突操作的SQL语句。但你必须考虑到查询优化器不得不去处理比示例中要复杂得多的SQL语句。试想一下，一个查询引用了几个视图而这些视图又引用了其他视图的情况，或者由通用用途的工具和专门的查询工具生成的查询语句。当查询优化器把所有的东西放在一起，出现多余或冲突操作的情况会屡见不鲜。此外，查询优化器能识别奇怪的情况并避免执行不必要的处理。还有，一些查询转换允许你按最自然、可读的方式书写SQL语句，而不用为性能牺牲清晰性。实际上，一些查询转换是非常常见的SQL优化技术，当手工应用时，会产生不易读的SQL语句。

6.3.1 计数转换

计数转换（Count Transformation）的目的是将count（列）表达式转化为count(*)。引入这个查询转换是因为相比count（列），处理count(*)时在索引使用方面有更多的选择空间。第13章会详细讨论这方面的内容。计数转换基于启发式的查询转换，可以在count函数引用的列中关联NOT NULL约束时进行应用（但是检查约束时不能用于此用途）。注意计数转换也会将count(1)这样的表达式转化为count(*)。

下面的例子基于count_transformation.sql脚本，阐明了这种查询转换。注意原始的查询包含count(n2)这个表达式：

```
SELECT count(n2)
FROM t
```

如果有一个NOT NULL约束定义在n2列上，计数转换就会将count(n2)转换为count(*)并生成以下查询：

```
SELECT count(*)
FROM t
```

6.3.2 公共子表达式消除

公共子表达式消除（common sub-expression elimination）的目的是移除重复的谓词从而避免多次处理同一个操作。这是一种基于启发式的查询转换。

下面的例子来自common_subexpr_elimination.sql这个脚本，注意两个分隔的谓词是如何重叠的。实际上，所有满足第一个条件的数据行都满足第二个条件：

```
SELECT *
FROM t
WHERE (n1 = 1 AND n2 = 2) OR (n1 = 1)
```

公共子表达式消除可移除冗余的谓词，并产生以下查询：

```
SELECT *
FROM t
WHERE n1 = 1
```

你是不是对留下的谓词感到惊讶？如果仔细考虑一下，会发现n1=1足以满足这个查询了，而n2=2却仍然需要考虑n1=1的情况。

6.3.3 “或”扩张

“或”扩张（or expansion）的目的是将查询的WHERE条件中包含分隔谓词的语句转化为使用一个或多个UNION ALL集合运算符的复合查询。通常情况下，每个分隔的谓词被转化成为一个组件查询。这里应用的是一个基于成本的查询转换，大多数时候是为了启用额外的索引访问路径。实际上，分隔的谓词和索引在一起搭配时并不总是进展顺利（参见第13章）。还要注意仅从11.2.0.2版本起这种查询转换才开始支持函数式索引。

注意 即使“或”扩张是基于成本的查询转换，查询优化器也会在尝试使用它之前检查一些启发式查询转换。如果查询转换不被允许，可能会错失一个拥有更低成本的执行计划。

接下来的例子基于or_expansion.sql这个脚本，注意，WHERE条件包含两个分隔的谓词。因为这些原因，查询优化器会评估一次基于表扫描的成本是否高于两次单独的基于索引扫描的成本：

```
SELECT pad
FROM t
WHERE n1 = 1 OR n2 = 2
```

如果两次索引扫描的成本更低，“或”扩张就会产生以下查询。注意，添加lnnvl(n1 = 1)这个谓词是为了避免多重记录。lnnvl函数在作为参数传递的条件为FALSE或NULL时返回TRUE。因此，第二个组件查询只会在第一个组件查询未返回某条记录的情况下才返回这条记录：

```
SELECT pad
FROM t
WHERE n1 = 1
UNION ALL
SELECT pad
FROM t
WHERE n2 = 2 AND lnnvl(n1 = 1)
```

一些分隔的谓词永远不会被显式地用“或”扩张转换。下面的查询就展示了这样一个例子。所有的谓词都引用了一个叫作n1的列，使得WHERE条件的内容能够像IN条件那样处理：

```
SELECT *
FROM t
WHERE n1 = 1 OR n1 = 2 OR n1 = 3 OR n1 = 4
```

6.3.4　视图合并

视图合并（View Merging）的目的是通过合并语句中一部分视图和内联视图，以便减少由它们产生的查询块的数量。引入这个查询转换的原因是，如果没有它，查询优化器就会分别处理每一个查询块。当分别处理每个查询块时，查询优化器无法保证每次都为整体SQL语句产生最优的执行计划。此外，由视图合并产生的查询块可能会进一步引导启用其他的查询转换。

查询块

简单地说，最顶级的SQL语句以及一个SQL语句中拥有自己的SELECT子句的每个扩展部分都是查询块。简单的SQL语句只有一个单独的查询块。而一旦使用了视图或者像子查询、内联视图以及集合运算符这样的结构，多重的查询块就出现了。例如，下面的查询有两个查询块（为了阐明问题，我用子查询分解子句代替定义一个真正的视图）。第一个查询块是顶层查询，就是引用dept表的那个查询。第二个查询块是使用WITH子句定义的查询，它引用的是emp表：

```
WITH emps AS (SELECT deptno, count(*) AS cnt
              FROM emp
              GROUP BY deptno)
SELECT dept.dname, emps.cnt
FROM dept, emps
WHERE dept.deptno = emps.deptno
```

视图合并有两个子范畴。

- **简单视图合并**（Simple view merging）用于合并简单的选择-投影-连接查询块[1]。因为它所处理情况的简单性，简单视图合并是一种基于启发式的查询转换。它无法应用于包含类似聚合、集合运算符、层次查询、MODEL子句或者SELECT列表中含有子查询这样的视图或内联视图。
- **复杂视图合并**（Complex view merging）用于合并包含聚合的查询块。这是一种基于成本的查询转换，无法应用于有层次查询出现或者包含GROUPING SETS、ROLLUP、PIVOT或者MODEL子句的视图或内联视图。

注意，因为应用复杂视图合并不一定能够带来好处，所以它是基于成本的查询转换。实际上，应用它时，物化视图或者内联视图中出现的聚合就被推后了，因此可能导致SQL在一个很大的结果集上执行。

[1] 选择-投影-连接查询块由三个基本操作组成：一个选择操作用于抽取满足指定谓词的记录，一个投影操作从引用的表中抽取指定的列，还有一个连接操作将不同表中抽取的数据放在一起。过滤和连接谓词基于类似等号这样的简单运算符。例如：SELECT t1.id, t2.n FROM t1 JOIN t2 ON t1.id = t2.id WHERE t1.n = 42。

视图合并可能会带来安全问题。为了预防这些问题，就提出了安全视图合并的概念，并由初始化参数optimizer_secure_view_merging控制其是否可用。第9章会详细讨论这个特性。

1. 简单视图合并

在下面这个来自simple_view_merging.sql脚本的例子中，查询由三个查询块构成：顶层查询和两个内联视图。注意，这两个内联视图是简单的选择-投影-连接查询块：

```
SELECT *
FROM (SELECT t1.*
      FROM t1, t2
      WHERE t1.id = t2.t1_id) t12,
     (SELECT *
      FROM t3
      WHERE id > 6) t3
WHERE t12.id = t3.t1_id
```

因为内联视图可以进行合并，简单视图合并产生了以下查询：

```
SELECT t1.*, t3.*
FROM t1, t2, t3
WHERE t1.id = t3.t1_id AND t1.id = t2.t1_id AND t3.id > 6
```

当涉及外链接时简单视图合并就不一定每次都能执行了。例如，在之前的查询中，如果把顶层查询的谓词改成t12.id = t3.t1_id(+)，视图合并仍可以执行，但是如果将谓词改成t12.id(+) = t3.t1_id就没法执行视图合并了。

2. 复杂视图合并

下面的例子来自complex_view_merging.sql脚本，展示了一个带有GROUP BY子句的内联视图。这样的查询按以下方式执行：访问内联视图中引用的表，评估GROUP BY子句和sum函数，最后将内联视图的结果集与顶层查询引用的表进行连接：

```
SELECT t1.id, t1.n, t1.pad, t2.sum_n
FROM t1, (SELECT n, sum(n) AS sum_n
          FROM t2
          GROUP BY n) t2
WHERE t1.n = t2.n
```

将GROUP BY子句的评估推迟直到连接完毕之后有利时，复杂视图合并产生以下查询：

```
SELECT t1.id, t1.n, t1.pad, sum(n) AS sum_n
FROM t1, t2
WHERE t1.n = t2.n
GROUP BY t1.id, t1.n, t1.pad, t1.rowid, t2.n
```

6.3.5 选择列表裁剪

选择列表裁剪（Select List Pruning）的目的是移除不必要的列或者移除来自子查询、内联视图以及普通视图的SELECT子句的表达式。这种类型的查询转换不会考虑顶层查询的SELECT子句。当一个列或者表达式没有在除引用或定义它的SELECT子句以外的地方被引用时，就会被认为是没有必要的。这是一种基于启发式的查询转换。

在下面这个来自select_list_pruning.sql脚本的例子中，注意，子查询引用的两个列（n2和n3）没有被外层的主查询引用：

```
SELECT n1
FROM (SELECT n1, n2, n3
      FROM t)
```

因为n2和n3这两个列是没有必要的，选择列表裁剪会移除它们并产生以下查询：

```
SELECT n1
FROM (SELECT n1
      FROM t)
```

使用视图合并，可以进一步简化查询。于是生成了下面这样的查询：

```
SELECT n1
FROM t
```

6.3.6　谓词下推

谓词下推（Predicate Push Down）的目的是将谓词下推到无法合并的视图或内联视图的内部。能够进行下推的谓词必须包含在拥有不可合并的视图或内联视图的查询块的内部。应用这种类型的查询转换有三个主要的原因：

- 为了启用额外的访问路径（典型的是索引扫描）；
- 为了启用额外的连接方法以及连接顺序；
- 为了确保能够尽可能快地应用谓词，从而避免不必要的处理操作。

谓词下推有两个子范畴：过滤谓词下推和连接谓词下推。两种变换的不同是由它们操作的谓词的类型决定的。

1. 过滤谓词下推

过滤谓词下推（Filter Push Down）的目的是将限制条件（过滤条件）下推到不可合并的视图或者内联视图的内部。这是一种基于启发式的查询转换。注意，这种查询转换不下推连接条件。下推连接条件是由下一节呈现的查询转换完成的。

下面的例子来自filter_push_down.sql脚本。UNION集合运算符用来防止内联视图与顶层查询合并：

```
SELECT *
FROM (SELECT *
      FROM t1
      UNION
      SELECT *
      FROM t2)
WHERE id = 1
```

过滤谓词下推将限制条件（id=1）下推到内联视图的内部并产生了下面的查询。现在，这两张表不仅可以通过索引来访问，同时也保证了UNION集合运算符需要的排序操作所处理的记录尽可能少：

```
SELECT *
FROM (SELECT *
      FROM t1
      WHERE id = 1
```

```
        UNION
        SELECT *
        FROM t2
        WHERE id = 1)
```

此外简单视图合并还消除了顶层查询块。

2. 连接谓词下推

连接谓词下推（Join Predicate Push Down）的目的是将连接谓词下推到无法合并的视图或内联视图的内部。这是一种基于成本的查询转换。

下面的例子来自join_predicate_push_down.sql脚本。UNION集合运算符用来防止内联视图与顶层查询合并。注意，内联视图与上一节例子中所用的是一样的（尽管表的名称不一样）。只不过在本例中，内联视图与另外一张表进行连接：

```
SELECT *
FROM t1, (SELECT *
          FROM t2
          UNION
          SELECT *
          FROM t3) t23
WHERE t1.id = t23.id
```

连接谓词下推将连接条件（t1.id = t23.id）下推到内联视图的内部并产生以下查询。这个查询享有与上一小节描述的查询相同的好处（启用了索引访问，减少了需要排序的数据总量）。在这个例子中，额外的访问路径也允许查询优化器自由选择所有可用的连接方法和连接顺序：

```
SELECT *
FROM t1, (SELECT *
          FROM t2
          WHERE t2.id = t1.id
          UNION
          SELECT *
          FROM t3
          WHERE t3.id = t1.id) t23
```

尽管前面的SQL语句并不是有效的（t1.id列在内联视图内部并不可见），但是SQL引擎可以处理与它类似的一些情况。为了支持这样的查询，从12.1版本开始可以使用侧向内联视图。例如，下面的查询在12.1版本中是合法的：

```
SELECT *
FROM t1, lateral(SELECT *
                 FROM t2
                 WHERE t2.id = t1.id
                 UNION
                 SELECT *
                 FROM t3
                 WHERE t3.id = t1.id) t23
```

6.3.7　谓词迁移

谓词迁移（Predicate Move Around）的目的是将限制条件（过滤条件）提取、跨越、下推到无法合并的视图或内联视图的内部。尽管这有点类似谓词下推，这个查询转换还能将谓词在彼此不包含的查询块之间移动。应用这种基于启发式的查询转换的主要原因是启用额外的访问路径（一般来说是索引扫描）并确保谓词能够尽可能快地应用。

下面例子中的两个内联视图来自redicate_move_around.sql脚本，因为DISTINCT运算符的原因不能与顶层查询合并。第一个内联视图在用于两个内联视图连接条件（t1.n = t2.n）的列（n）上有一个限制条件：

```
SELECT t1.pad, t2.pad
FROM (SELECT DISTINCT n, pad
      FROM t1
      WHERE n = 1) t1,
     (SELECT DISTINCT n, pad
      FROM t2) t2
WHERE t1.n = t2.n
```

在本例中，谓词迁移执行以下三个主要步骤。

(1) 将限制条件（n = 1）从第一个内联视图中提取到顶层查询中。

(2) 在限制条件（t1.n = 1）与连接条件（t1.n = t2.n）之间应用传递特性，从而产生新的谓词（t2.n = 1）。

(3) 将新的谓词下推到第二个内联视图内部。

结果就是谓词迁移生成了接下来的查询语句。将谓词添加到第二个内联视图内部，不仅允许通过索引访问t2表，而且也相应地减少了DISTINCT运算符需要处理的数据总量：

```
SELECT t1.pad, t2.pad
FROM (SELECT DISTINCT n, pad
      FROM t1
      WHERE n = 1) t1,
     (SELECT DISTINCT n, pad
      FROM t2
      WHERE n = 1) t2
WHERE t1.n = t2.n
```

6.3.8　非重复放置

非重复放置（Distinct Placement）的目的是尽快消除重复。这种基于成本的查询转换仅从11.2版本开始起才可用。

DISTINCT运算符的作用是从结果集中去除重复。当它在查询中与其他一个或多个联接一同被指定时，从概念上讲，数据库引擎应该在联接处理完毕后再处理DISTINCT运算符。然而，要达到最佳性能，在某些情况下需要在处理联接之前消除重复。更早消除重复可以保证中间结果集尽可能小，这样需要联接处理的数据也随之减少了。

下面的例子来自distinct_placement.sql脚本，在两张表中存在着父-子关系。进一步来说，你可以假设子表（t2）比父表（t1）包含更多的记录：

```
SELECT DISTINCT t1.n, t2.n
FROM t1, t2
WHERE t1.id = t2.t1_id
```

当t2.n的不同值的数量远远少于子表中存储的数量时，非重复放置就产生了下面的查询。注意额外增加的DISTINCT运算符，它应用于子表的数据，在联接父表之前就消除了重复：

```
SELECT DISTINCT t1.n, vw_dtp.n
FROM t1, (SELECT DISTINCT t2.t1_id, t2.n
          FROM t2) vw_dtp
WHERE t1.id = vw_dtp.t1_id
```

6.3.9 非重复消除

从10.2.0.4版本开始可用的非重复消除（Distinct Elimination）的目的，是移除对于保证结果集不包含重复数据来说不需要的DISTINCT运算符。这是一种基于启发式的查询转换，可以在SELECT子句涉及以下情况时使用，在不修改要查询的列的情况下，查询所有的主键列、所有非空的唯一键列或rowid。

下面的例子来自distinct_elimination.sql脚本。注意，表的主键定义在名为id的列上：

```
SELECT DISTINCT id, n
FROM t
```

由于id列的出现，也就是表的主键，在SELECT子句中就足以保证数据行的唯一性。于是，非重复消除产生了以下查询：

```
SELECT id, n
FROM t
```

6.3.10 Group-by 放置

Group-by放置（Group-by Placement）的目的基本上与非重复放置一样。唯一明显的差别是它们应用的查询类型不同。前者适用于包含GROUP BY子句的查询，后者适用于包含DISTINCT运算符的查询。Group-by放置是基于成本的查询转换，从11.1版本开始起才可用。

在下面这个来自group_by_placement.sql脚本的例子中，在两张表中存在父–子关系，且子表（t2）包含的记录数比父表（t1）更多：

```
SELECT t1.n, t2.n, count(*)
FROM t1, t2
WHERE t1.id = t2.t1_id
GROUP BY t1.n, t2.n
```

当t2.n中不同值的数量远远小于子表中存储的数据行的数量时，group-by放置就产生了下面的查询。一个额外的GROUP BY子句被应用于子表的数据上，以便在连接父表之前消除重复。还要注意，在顶层查询的SELECT子句中，count函数被sum函数替换了：

```
SELECT t1.n, vw_gb.n, sum(vw_gb.cnt)
FROM t1, (SELECT t2.t1_id, t2.n, count(*) AS cnt
          FROM t2
          GROUP BY t2.t1_id, t2.n) vw_gb
WHERE t1.id = vw_gb.t1_id
GROUP BY t1.n, vw_gb.n
```

6.3.11 Order-By 消除

Order-By消除（Order-By Elimination）的目的是从子查询、内联视图以及常规视图中移除不必要的ORDER BY子句。很显然这种基于启发式的查询转换不会考虑顶层SELECT子句。当一个ORDER BY后面跟随一个不保证会按顺序返回数据的操作时，或者跟随一个会按照不同顺序返回数据的操作时，就可以认为这个ORDER BY子句是没有必要的；例如，后面跟随另一个ORDER BY或者聚合。

在下面这个来自order_by_elimination.sql脚本的例子中，不仅内联视图中有一个ORDER BY，而且顶层查询块中还有一个GROUP BY：

```
SELECT n2, count(*)
FROM (SELECT n1, n2
      FROM t
      ORDER BY n1)
GROUP BY n2
```

因为顶层查询块中的GROUP BY并不保证按顺序返回数据，所以order-by消除就会移除ORDER BY并产生以下查询：

```
SELECT n2, count(*)
FROM (SELECT n1, n2
      FROM t)
GROUP BY n2
```

查询优化器使用选择列表裁剪和简单视图合并进一步将查询转化成以下形式：

```
SELECT n2, count(*)
FROM t
GROUP BY n2
```

6.3.12 子查询展开

子查询展开（Subquery Unnesting）的目的是将半连接（IN, EXISTS）、反连接（NOT IN, NOT EXISTS）以及标量子查询注入到查询块包含的FROM子句中去，并将它们转化为内联视图。一些展开是由基于启发式的查询转换执行的，另外一些则是由基于成本的查询转换实施的。应用这种查询转换的主要原因是启用所有可能的连接方法。事实上，如果没有子查询展开，子查询可能就不得不在包含它的查询块每返回一行数据时都执行一遍（详见第10章）。然而子查询展开也并非总能执行。例如，如果子查询中包含某种类型的聚合或者包含rownum伪列，展开都不可能执行。当半连接和反连接子查询中含有集合运算符时，只有从11.2版本开始才可以展开。此外，从12.1版本开始，标量子查询展开改进为在SELECT子句中处理标量子查询。

下面的例子来自subquery_unnesting.sql脚本，演示了这种查询转换是如何工作的：

```
SELECT *
FROM t1
WHERE EXISTS (SELECT 1
              FROM t2
              WHERE t2.id = t1.id
              AND t2.pad IS NOT NULL)
```

子查询展开可以归纳为两个步骤。第一步，如下面的查询所示，重写子查询为内联视图。注意，下面展示的并不是合法的SQL语句，因为实现半连接的运算符（s=）在SQL语法中并不可用（只在SQL引擎内部使用）：

```
SELECT *
FROM t1, (SELECT id
          FROM t2
          WHERE pad IS NOT NULL) sq
WHERE t1.id s= sq.id
```

第二步，正如此处展示的，将内联视图重写为正常的连接：

```
SELECT t1.*
FROM t1, t2
WHERE t1.id s= t2.id AND t2.pad IS NOT NULL
```

尽管前面的例子是基于半连接的，这个查询转换同样适用于反连接。唯一的区别是使用反连接运算符（a=），而不是半连接运算符（s=）。这是另外一个SQL引擎仅在内部使用的运算符。

6.3.13 子查询合并

子查询合并（Subquery Coalescing）的目的是将等价的半连接以及反连接子查询组合到同一个查询块中。应用这种自11.2版本起可用的基于启发式的查询变换，其主要目的是减少表访问的数量，从而减少连接的数量。

下面的例子来自subquery_coalescing.sql脚本，演示了这种查询转换是如何工作的。注意，两个相互关联的子查询处理相同的数据，只是限制条件有所区别：

```
SELECT *
FROM t1
WHERE EXISTS (SELECT 1
              FROM t2
              WHERE t2.id = t1.id AND t2.n > 10)
OR EXISTS (SELECT 1
           FROM t2
           WHERE t2.id = t1.id AND t2.n < 100)
```

子查询合并组合两个子查询并产生以下查询：

```
SELECT *
FROM t1
WHERE EXISTS (SELECT 1
              FROM t2
              WHERE t2.id = t1.id AND (t2.n > 10 OR t2.n < 100))
```

通过子查询展开，可以将查询进一步转化成以下形式。注意，正如在上一节中所解释的，下面的查询使用一个特殊的运算符（s=）来实现半连接。这个运算符仅可以在SQL引擎内部使用，并非那种可以在书写SQL语句时指定的语法的一部分：

```
SELECT t1.*
FROM t1, t2
WHERE t1.id s= t2.id AND (t2.n > 10 OR t2.n < 100)
```

6.3.14　使用窗口函数移除子查询

使用窗口函数移除子查询（Subquery Removal Using Window Function）的目的是使用窗口函数替换包含聚合函数的子查询。这是一种基于启发式的查询转换，可以在一个查询块包含所有出现在一个子查询中的表和谓词时应用。

下面的例子基于subquery_removal.sql脚本，演示了这种查询转换是如何工作的。注意，在顶层查询和子查询中都引用了t2表：

```
SELECT t1.id, t1.n, t2.id, t2.n
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t2.n = (SELECT max(n)
            FROM t2
            WHERE t2.t1_id = t1.id)
```

查询转换移除子查询，并产生接下来的查询。注意CASE表达式是如何用来生成某种标志的。这种标志用来识别那些满足在原始查询中由子查询指定的限制条件的记录：

```
SELECT t1_id, t1_n, t2_id, t2_n
FROM (SELECT t1.id AS t1_id, t1.n AS t1_n, t2.id AS t2_id, t2.n AS t2_n,
             CASE t2.n
              WHEN max(t2.n) OVER (PARTITION BY t2.t1_id) THEN 1
            END AS max
      FROM t2, t1
      WHERE t1.id = t2.t1_id) vw_wif
WHERE max IS NOT NULL
```

6.3.15　联接消除

联接消除（Join Elimination）的目的是移除冗余的联接，换句话说，是为了在即使SQL语句明确要求的情况下也能够避免执行联接。对于查询优化器来讲，决定实现这种查询转换是否合理的关键信息，是外键的可用性是强制的还是被标记为RELY的。此外，从11.2版本开始，还会将基于主键的自联接纳入考虑范围之内。这种基于启发式的查询转换在使用包含联接的视图时尤其有用。注意，无论如何，联接消除也可以应用于没有视图的SQL语句。

来看下面这个基于join_elimination.sql脚本的例子。下面的SQL语句定义了一个视图。注意，在这两张表之间存在着父-子关系。实际上是t2表用它的t1_id列引用了t1表的主键：

```
CREATE VIEW v AS
SELECT t1.id AS t1_id, t1.n AS t1_n, t2.id AS t2_id, t2.n AS t2_n
FROM t1, t2
WHERE t1.id = t2.t1_id
```

当执行简单的SELECT * FROM v语句时，可以执行简单视图合并，然后这个查询就会转化如下：

```
SELECT t1.id AS t1_id, t1.n AS t1_n, t2.id AS t2_id, t2.n AS t2_n
FROM t1, t2
WHERE t1.id = t2.t1_id
```

但是，如下一个例子所演示的，仅当引用在子表中定义的列时（例如，SELECT t2_id, t2_n FROM v），查询优化器才能消除与父表的联接。这种转换能够实现是因为，外键约束保证t2表中所有的记录

一定引用t1表中的一条记录且只引用一条：

```
SELECT t2.id AS t2_id, t2.n AS t2_n
FROM t2
```

6.3.16 联接因式分解

这种自11.2版本开始可用的联接因式分解（Join Factorization）的目的，是识别出正在处理的联合查询的一部分是否可以在各个组成查询中共享，进而避免重复的数据访问和联接。实际上，没有这种查询转换，所有的组件查询在应用集合运算符之前都得单独执行。这是一种基于成本的查询转换，查询优化器只有在基于UNION ALL集合运算符的联合查询中会应用它。

下面的例子来自join_factorization.sql脚本。注意这两个组件查询不仅是访问同一张表，它们还都在相同的表（t2）中施加了一个限制条件。没有这种查询转换，两个组件查询都会单独执行，两个查询中的表都会被访问两次：

```
SELECT *
FROM t1, t2
WHERE t1.id = t2.id AND t2.id < 10
UNION ALL
SELECT *
FROM t1, t2
WHERE t1.id = t2.id AND t2.id > 990
```

为避免重复处理访问每张表两次的工作，联接因式分解可以转换这个查询，如下面的例子所示。因为表t1被因式分解了，所以它只需访问一次。根据表的大小以及所选择的用于从中抽取数据的访问路径的不同，在I/O和CPU使用方面节省的成本可能会非常显著：

```
SELECT t1.*, vw_jf.*
FROM t1, (SELECT *
          FROM t2
          WHERE id < 10
          UNION ALL
          SELECT *
          FROM t2
          WHERE id > 990) vw_jf
WHERE t1.id = vw_jf.id
```

6.3.17 外联接转内联接

外联接转内联接（Outer Join to Inner Join）的目的是将不必要的外联接转化为内联接。这样做是因为外联接可能会阻止查询优化器选择某种特定的联接方法或联接顺序。这是一种基于启发式的查询转换。

下面的例子基于outer_to_inner.sql脚本，演示了这种查询转换。注意，限制条件（t2.id IS NOT NULL）与外联接条件（t1.id = t2.t1_id(+)）有冲突：

```
SELECT *
FROM t1, t2
WHERE t1.id = t2.t1_id(+) AND t2.id IS NOT NULL
```

查询转换移除了外联接运算符以及多余的谓词，并产生了以下查询：

```
SELECT *
FROM t1, t2
WHERE t1.id = t2.t1_id
```

6.3.18　完全外联接

完全外联接（Full Outer Join）是一种基于启发式的查询转换，其目的是将完全外联接转换为一个使用UNION ALL集合运算符的复合查询，从而组合由外联接和反联接返回的记录。此外， 如果ON子句指定的谓词引用了非空列，并且列上定义了强制的或标记为RELY的外键约束，查询转换甚至能够将完全外联接转换成一个会在运行时作为左外联接执行的查询语句。

注意　尽管完全外联接的语法从9.0版本开始就可用，但是在11.1版本之前SQL引擎没有能力执行原生的完全外联接。因此，会将包含完全外联接的查询转换成某种SQL引擎能够工作的变体。自11.1版本开始，随着原生完全外联接的引入，这种情况就不存在了。

下面的例子来自full_outer_join.sql脚本，演示了这种查询转换是如何工作的。注意，这个查询在FROM子句中使用了FULL OUTER JOIN语法：

```
SELECT *
FROM t1 FULL OUTER JOIN t2 ON t1.n = t2.n
```

查询转换产生了下面的查询。注意，在内联视图中定义的两个联接中，只有第一个是外联接。第二个联接使用之前在6.3.12节中解释的特殊运算符（a=）来实现反联接：

```
SELECT id1 AS id, n1 AS n, pad1 AS pad, id, t1_id, n, pad
FROM (SELECT t1.id AS id1, t1.n AS n1, t1.pad AS pad1, t2.id, t2.t1_id, t2.n, t2.pad
      FROM t1, t2
      WHERE t1.n = t2.n(+)
      UNION ALL
      SELECT NULL, NULL, NULL, t2.id, t2.t1_id, t2.n, t2.pad
      FROM t1, t2
      WHERE t1.n a= t2.n) vw_foj
```

6.3.19　表扩张

自11.2版本开始，可用的表扩张（Table Expansion）的目的是通过利用部分不可用的索引尽可能多地利用索引扫描。最关键的是要认识到，这种基于成本的查询转换只有满足以下三个基本条件才会考虑应用：

- 涉及了一个分区表；
- 这张分区表有部分分区不可用的本地索引，或者从12.1开始，有一个局部索引；
- 要进行优化的SQL语句不得不同时处理可用和不可用索引分区所覆盖的数据。

在11.2版本之前遇到这种情况时，基于部分不可用的索引的索引扫描是不可能的，所以整个索引会被完全忽略掉，从而不得不使用全表扫描。但是通过表扩张，当可用的索引分区出现时，查询优化器能够利用它们，并且在遇到不可用索引分区时，会回退到全分区扫描。

在下面这个来自table_expansion.sql脚本的例子中，有一个范围分区表，该表将2014年的每个季度作为一个分区。注意，它建立的本地索引是不可用的。稍后会为一个单独的表分区建立一个可用的索引分区。

```
CREATE TABLE t (
  id NUMBER PRIMARY KEY,
  d DATE NOT NULL,
  n NUMBER NOT NULL,
  pad VARCHAR2(4000) NOT NULL
)
PARTITION BY RANGE (d) (
  PARTITION t_q1_2014 VALUES LESS THAN (to_date('2014-04-01','yyyy-mm-dd')),
  PARTITION t_q2_2014 VALUES LESS THAN (to_date('2014-07-01','yyyy-mm-dd')),
  PARTITION t_q3_2014 VALUES LESS THAN (to_date('2014-10-01','yyyy-mm-dd')),
  PARTITION t_q4_2014 VALUES LESS THAN (to_date('2015-01-01','yyyy-mm-dd'))
);

CREATE INDEX i ON t (n) LOCAL UNUSABLE;

ALTER INDEX i REBUILD PARTITION t_q4_2014;
```

在11.2版本之前，优化下面这样的查询时，会完全避开索引扫描。即使限制条件是基于索引列的，也并非所有需要的索引分区都是可用的。于是，即使有可用索引的表分区，也不会执行索引扫描。

```
SELECT *
FROM t
WHERE n = 8
```

自11.2版本开始，为了能够利用可用的索引分区，会将查询转换成一个联合查询，其中一个组件查询访问带有可用索引的分区，另一个组件查询访问带有不可用索引的分区。注意这种转换是如何通过向两个组件查询都加入基于分区键的谓词来实现的：

```
SELECT *
FROM (SELECT *
      FROM t
      WHERE n = 8
      AND d < to_date('2014-10-01','yyyy-mm-dd')
      UNION ALL
      SELECT *
      FROM t
      WHERE n = 8
      AND d >= to_date('2014-10-01','yyyy-mm-dd')
      AND d < to_date('2015-01-01','yyyy-mm-dd')) vw_te
```

6.3.20 集合操作联接转变

集合操作联接转变（Set to Join Conversion）的目的是在涉及INTERSECT和MINUS的联合查询中避免排序操作。这种查询转换还为这样的查询将消除重复推迟到处理工作的最后。

基于INTERSECT和MINUS集合运算符的联合查询基本上按以下两个步骤执行。

(1) 每个组件查询独立执行，然后将结果集排序，并消除重复记录。

(2) 接下来执行集合运算符，然后最终结果集就确定了。

这种涉及INTERSECT或MINUS运算的查询方式并非总是高效的。举例来说，当组件查询返回大量数据，但这些数据中的大部分被集合运算符所消除时，大部分后来被消除的数据最后都进行了不必要的排序。通过将查询转换为一种允许在排序之前就丢弃冗余数据的方式，集合操作联接转变可以避免这种低效运算。此外，因为集合运算符被联接取代了，也启用了额外的访问路径。这是一种基于启发式的的查询转换，默认情况下并未启用[①]。为了利用这种转换必须使用hint set_to_join。

下面的例子来自set_to_join.sql脚本。这个联合查询基于INTERSECT集合运算符：

```
SELECT *
FROM t1
WHERE n > 500
INTERSECT
SELECT *
FROM t2
WHERE t2.pad LIKE 'A%'
```

查询转换将集合运算符转化成了一个联接。此外，为确保仅返回需要的记录，查询转换还添加了几个谓词和一个DISTINCT运算符。下面是集合操作联接转变完成以后最终的查询语句：

```
SELECT DISTINCT t1.*
FROM t1, t2
WHERE t1.id = t2.id AND t1.n = t2.n AND t1.pad = t2.pad
AND t1.n > 500 AND t1.pad LIKE 'A%'
AND t2.n > 500 AND t2.pad LIKE 'A%'
```

6.3.21　星型转换

星型转换（Star Transformation）是一种基于成本的查询转换，用于从星型模型中提取数据的查询。第14章将介绍关于星型模型和针对这种模型的查询优化的详细信息。

6.3.22　物化视图查询重写

物化视图查询重写（Query Rewrite with Materialized View）是一种优化技术。该技术允许数据库引擎在即使待优化的查询并未直接引用物化视图的情况下，也能访问在物化视图中存储的数据。第15章详细讨论了这种类型的查询转换。

6.4　小结

本章描述了关于查询优化器的基础知识。你已经了解了查询优化器用于生成执行计划的输入项的内容，也学习了构成SQL引擎的关键组件以及它们彼此之间是如何相互作用的，同时学习了关于查询转换的内容，以及如何应用它们来增加为某个给定SQL语句找寻更好执行计划的机会。

系统统计信息是本章介绍的输入项中的一种。它们的用途是描述运行数据库引擎的系统以及存储子系统在运行时的表现。第7章将详细描述什么是系统统计信息，如何管理它们，以及查询优化器如何利用它们增强估算能力。

① 初始化参数_convert_set_to_join默认设置为FALSE。

第 7 章

系统统计信息

查询优化器曾基于执行SQL语句需要的物理读的数量进行成本估算。这一方法称为I/O成本模型。该方法的主要缺陷是，单块物理读和多块物理读在成本上是等价的[1]。结果，类似全表扫描这样的多块读操作受到了青睐。在引入系统统计信息之前，尤其是在OLTP系统上，optimizer_index_caching和optimizer_index_cost_adj这两个初始化参数曾用于解决这个问题（详见第9章）。事实上， 这两个参数的默认值过去只适合报表系统和数据仓库。如今，一种新的成本计算方法，即CPU成本模型，用于纠正这一缺陷。要使用CPU成本模型，必须将系统统计信息（即，关于数据库引擎运行所在系统的性能的额外信息）提供给查询优化器。从本质上讲，系统统计信息提供以下信息：

- 磁盘I/O子系统的性能
- CPU的性能

尽管其名称如此，CPU成本模型同样考虑到了物理读。而且也考虑到了磁盘I/O子系统的性能表现，而不是仅仅把I/O成本建立在物理读数值的基础上。不要让名称误导你。

通常总是有一组默认的系统统计信息可供使用。因此，默认情况下CPU成本模型是启用的。实际上，使用I/O成本模型的唯一途径是在SQL语句级别指定no_cpu_costing 这个hint，或者设置一个未公开的初始化参数。在其他所有情况下，查询优化器都使用CPU成本模型。

7.1 dbms_stats 包

dbms_stats包提供一组全面的存储过程来管理系统统计信息。默认情况下，这个包直接修改数据字典。尽管如此，包中大部分存储过程都可以使用一张存储在数据字典之外的用户自定义表进行运转。这张表就是我所说的备份表。该表主要用于以下两种情形。

- 在不必将统计信息存储在数据字典中的情况下收集系统统计信息，因此也就无需在收集完毕后使它们立即对查询优化器可见。
- 为了在两个数据库之间移动系统统计信息。

在两个数据库之间移动系统统计信息就是，将系统统计信息导出到源端数据库的一张备份表中，将这张备份表移动到另一个数据库，然后将表中的内容导入到目标数据字典中。在数据库之间移动系统统计信息是一种确保所有与某一给定的应用程序（例如，开发、测试和生产环境）有关的数据库都

[1] 一般来说，读单个块比读多个快要快些。但说来奇怪，在现实中并非总是这样的。在任何情况下，重要的是，要记住这两者之间确实有所区别。

使用相同的系统统计信息方法。

注意 系统统计信息的收集不会使存储在库缓存中的游标失效。因此，新的系统统计信息只会作用于即将需要进行硬解析的SQL语句。

---

dbms_stats包提供以下子程序（见图7-1）：

- gather_system_stats收集系统统计信息并将它们存储在数据字典或备份表中；
- delete_system_stats删除存储在数据字典或备份表中的系统统计信息；
- restore_system_stats将系统统计信息还原到数据字典中；
- export_system_stats将系统统计信息从数据字典移动到备份表中；
- import_system_stats将系统统计信息从备份表移动到数据字典中；
- get_system_stats提取存储在数据字典或备份表中的系统统计信息；
- set_system_stats修改存储在数据字典或备份表中的系统统计信息。

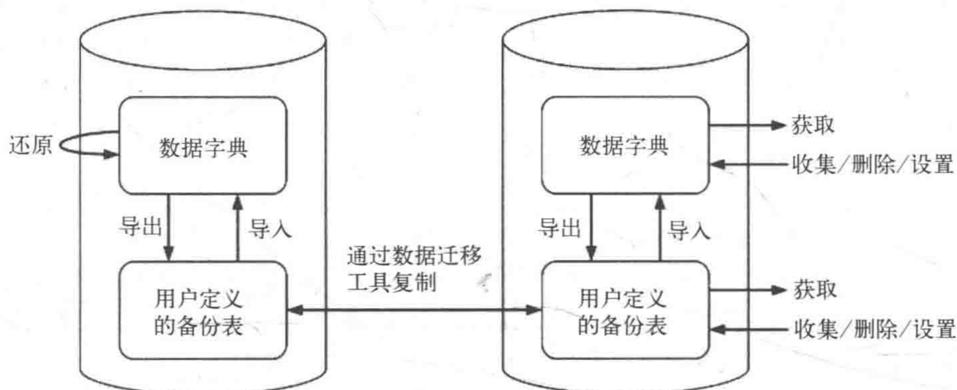


图7-1 dbms_stats包提供一组全面的存储过程来管理系统统计信息

默认情况下，执行dbms_stats包的权限是授予public的。这就意味着每个用户都可以收集系统统计信息。尽管如此，只有那些持有gather_system_statistics角色所提供权限的用户才能将系统统计信息存储到数据字典中。没有授权的用户只能将它们存储到备份表中。默认情况下，gather_system_statistics角色是通过dba角色提供的。

7.2 有哪些系统统计信息可用

有两种类型的系统统计信息：*无负载统计信息*和*工作负载统计信息*。这两种统计信息的主要区别就是用于测量磁盘I/O子系统的的方法不同。前者运行复合基准测试，而后者使用应用基准测试。两种情况下CPU的性能都是通过复合基准测试计算的。在深入讨论这两种方法的区别之前，我们先来看看系统统计信息在数据字典中是如何存储的。

应用基准测试与复合基准测试

应用基准测试，也称为真实基准测试，它基于真实应用程序的日常操作产生的负载。尽管它通常可以非常精确地提供关于运行应用的系统的真实性能数据，但由于其本身的特性，以可控的方式应用它并非总是可行的。

复合基准测试是由不执行真实工作的程序所产生的工作负载。总体思路是它应该通过执行类似的操作模拟应用程序的负载。尽管这种方法可以很容易通过可控的方式进行应用，但是通常它不会产生与应用基准测试一样精确的性能指标。虽然如此，它在对比不同系统方面仍非常有用。

系统统计信息存储在字典表aux_stats$[1]中。然而，没有一个数据字典视图能够使其具体化。在这张表中，由下面三组通过sname列的不同值来区分的数据集合来记录。

- 值为SYSSTATS_INFO的记录是包含系统统计信息状态以及对应收集时间的数据集合。如果正确收集了这些信息，则会将STATUS列设置为COMPLETED。如果在收集统计信息过程中出现了问题，则会将STATUS列设置为BADSTATS，也就是说查询优化器不会使用系统统计信息。在收集工作负载统计信息时可能还会看见另外两个值：MANUALGATHERING和AUTOGATHERING。名为FLAGS的属性接受以下三个值：0代表系统统计信息通过调用delete_system_stats存储过程设置为默认值；1代表系统统计信息是正常收集或设置的；128表示系统统计信息通过调用restore_system_stats存储过程进行还原。

```
SQL> SELECT pname, pval1, pval2
  2  FROM sys.aux_stats$
  3  WHERE sname = 'SYSSTATS_INFO';

PNAME    PVAL1 PVAL2
------- ------ -----------------
DSTART         10-25-2013 23:26
DSTOP          10-25-2013 23:28
FLAGS        1
STATUS         COMPLETED
```

- 值为SYSSTATS_MAIN的记录是包含系统统计信息本身的数据集合。详细信息会在下节中介绍。

```
SQL> SELECT pname, pval1
  2  FROM sys.aux_stats$
  3  WHERE sname = 'SYSSTATS_MAIN';

PNAME              PVAL1
----------- ------------
CPUSPEEDNW        1991.0
IOSEEKTIM           10.0
IOTFRSPEED        4096.0
SREADTIM             1.6
MREADTIM             7.8
CPUSPEED          1992.0
MBRC                21.0
MAXTHR       659158016.0
SLAVETHR      34201600.0
```

[1] 如果无法直接访问字典表awx_stats$，可转而使用get_system_stats程序。

- 值为SYSSTATS_TEMP的记录是包含用于计算系统统计信息的值的数据集合。只在收集工作负载统计信息时可见。

7.3 收集系统统计信息

正如刚刚所描述的，数据库引擎支持两种类型的系统统计信息：无负载统计信息和工作负载统计信息。本节不仅会介绍它们是如何收集的，而且还会说明它们向查询优化器提供什么样的信息，并会说明如何决定应该采用无负载统计信息还是工作负载统计信息。

因为对于单个数据库来讲只存在一组单独的统计信息，所以RAC系统中的所有实例都使用相同的系统统计信息。因此，如果各个节点大小或负载不均衡，就必须仔细判断应该收集哪个节点的系统统计信息。

7.3.1 无工作负载统计信息

无工作负载统计信息总是可用的。如果显式删除它们，那么它们会在数据库下次启动过程中自动收集。因为数据库引擎采用了复合基准来产生用于测量系统性能的负载，所以可以在空闲的系统上收集无工作负载统计信息。为了测量CPU的速度，很可能会循环执行某种标准化的操作。为了测量磁盘I/O性能，会在数据库的几个数据文件上执行一些不同大小的读操作。

要收集无工作负载统计信息，需要将gather_system_stats存储过程的gathering_mode参数设置为noworkload，如下例所示：

```
dbms_stats.gather_system_stats(gathering_mode => 'noworkload')
```

此外，为了更好地支持拥有更高磁盘I/O吞吐量的系统（例如Exadata），从11.2.0.4版本（或者是安装了与bug 0248538相关的增强的补丁）开始有了另外一种收集无工作负载统计信息的方法：将gathering_mode参数的值设置为exadata，如下所示：

```
dbms_stats.gather_system_stats(gathering_mode => 'exadata')
```

这两种情况下，收集过程通常持续几分钟，然后就会计算出表7-1所列举的统计信息。奇怪的是，有时候需要重复收集统计信息的过程；否则，就会使用表7-1中的默认值。这是因为测量出来的统计信息在存储之前必须通过合理性检查。如果它们通不过合理性检查，就会被丢弃并被默认的统计信息替换掉。然而，进行这些检查时没有可以提供给你的信息。

表7-1 存储在数据词典中的无工作负载统计信息

名　称	描　述
CPUSPEEDNW	CPU每秒钟能够处理的操作数量（单位为百万次）。因为CPUSPEEDNW总是基于用来评估CPU速度的复合基准的结果，所以没有默认值
IOSEEKTIM	定位磁盘数据所需平均时间（单位为毫秒，平均寻道时间）。默认值是10
IOTFRSPEED	每毫秒能够从磁盘传输的平均字节数。默认值是4096
MBRC	多块读操作每次读的块的数量。该统计信息只能在exadata模式下设置（即，将gathering_mode设置为exadata）。MBRC没有默认值，因为它总是与db_file_multiblock_read_count初始化参数一致

普通无工作负载统计信息与exadata无工作负载统计信息的唯一区别就是后者的mbrc统计信息也

被设置了。更确切地说，它被设置成db_file_multiblock_read_count初始化参数的取值。其目的是告知查询优化器数据库引擎能够高效地执行大型的磁盘I/O操作，因此可以降低全表扫描的成本。只有在db_file_multiblock_read_count初始化参数没有明确设置的情况下才有必要使用exadata收集模式。没有设置这个参数时，查询优化器会使用8这个值来计算全表扫描的成本（详见第9章）。使用这样的值，全表扫描的成本通常会比预期成本要高出许多。使用exadata模式时，情况就完全不一样了。实际上，大多数系统上都会将mbrc设置为128，因此全表扫描的成本要低出许多。在拥有很高的磁盘I/O吞吐量的系统（例如Exadata）上使用exadata模式尤为明智。

7.3.2 工作负载统计信息

只有进行了明确的收集操作，工作负载统计信息方才可用。你不能使用一个空闲的系统来收集它们，因为数据库引擎需要利用日常数据库负载来衡量磁盘I/O子系统的性能。另一方面，用于无负载统计信息的方法可用来衡量CPU的速度。如图7-2所示，收集工作负载统计信息是一个三步操作的活动。收集的思路是要计算一个操作花费的平均时间，因此，有必要知道这个操作执行过多少次，以及有多少时间花费在执行这个操作上面。例如，使用下面的SQL语句，我能够计算出测试数据库的平均单块读的时间（6.2毫秒），这与dbms_stats包的执行方式一样：

```
SQL> SELECT sum(singleblkrds) AS count, sum(singleblkrdtim)*10 AS time_ms
  2  FROM v$filestat;

     COUNT    TIME_MS
---------- ----------
     22893      36760

SQL> REMARK run a benchmark to generate some disk I/O operations...

SQL> SELECT sum(singleblkrds) AS count, sum(singleblkrdtim)*10 AS time_ms
  2  FROM v$filestat;

     COUNT    TIME_MS
---------- ----------
     54956     236430

SQL> SELECT round((236430-36760)/(54956-22893),1) AS avg_tim_singleblkrd
  2  FROM dual;

AVG_TIM_SINGLEBLKRD
-------------------
                6.2
```

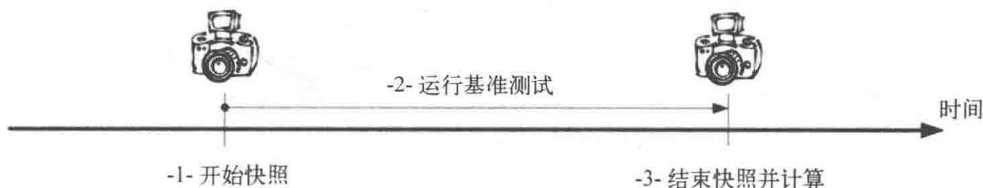


图7-2 为收集（计算）系统统计信息，使用了一些性能指标的两个快照

图7-2中列举的步骤如下。

(1) 一个含有若干性能数据的快照被捕获下来并存储在aux_stats$数据字典表中（对于这些数据，sname列的值被设置为SYSSTATS_TEMP）。这个步骤是通过将gather_system_stats存储过程的gathering_mode参数设置为start来执行的，如下面的命令所示：

```
dbms_stats.gather_system_stats(gathering_mode => 'start')
```

(2) 数据库引擎并不控制数据库的负载。这样会导致在下一次捕获快照之前，必须等待足够长的时间才能涵盖一个有代表性的负载。很难给出关于等待时间的通用建议，但是一般情况下至少应该等待5~10分钟。

(3) 捕获第二个快照。这一步骤是通过将gather_system_stats存储过程的gathering_mode参数设置为stop来实现的，如下面的命令所示：

```
dbms_stats.gather_system_stats(gathering_mode => 'stop')
```

(4) 基于两次快照的性能统计数据，计算表7-2中所列的系统统计信息。如果其中的一个磁盘IO统计信息无法计算，那么就会将这个统计信息设置为NULL。如果工作负载没有使用单块读、多快读或者并行处理中的一种，那么可能会出现无法计算某种统计信息的情况。例如，如果工作负载没有执行任何的多块读，则会将mbrc和mreadtim设置为NULL。

表7-2　存储在数据字典中的工作负载统计信息

列　名	描　述
CPUSPEED	一个CPU每秒钟能够处理的操作数量（单位为百万次）
SREADTIM	执行一个单块读操作所需的平均时间（单位为毫秒）
MREADTIM	执行一个多块读操作所需的平均时间（单位为毫秒）
MBRC	多块读操作过程中读取的平均块数量
MAXTHR	整个系统的最大磁盘I/O吞吐率（以字节每秒为单位）
SLAVETHR	一个单独的并行处理子进程的平均磁盘I/O吞吐率（以字节每秒为单位）

为避免手工捕获结束的快照，也可以将gather_system_stats存储过程的参数gathering_mode设置为interval。使用此参数的情况下，起始的快照会立即进行捕获，而结束的快照会安排在第二个名为interval的参数指定的分钟数之后执行捕获。下面的命令指定收集统计信息的过程持续10分钟：

```
dbms_stats.gather_system_stats(gathering_mode => 'interval',
                               interval       => 10)
```

注意，上面的命令的执行并不会花费10分钟。它只是捕获起始的快照，并安排一个在10分钟后捕获结束的快照的任务 。可以通过查询看见这个任务，例如user_scheduler_jobs视图。

警告　因为bug 9842771，工作负载系统统计信息，尤其是sreadtim和mreadtim的值，在11.2.0.1版本和11.2.0.2版本中是有缺陷的。要修复这个问题，你可以安装补丁9842771。如果无法安装这个补丁，那么可以采用的解决方案是，手工设置sreadtim和mreadtim的值（稍后的代码样例中会介绍如何设置这些值）。

收集工作负载统计信息的主要问题是选择合适的收集时长。事实上，大多数的系统都经历着除了恒定负载以外的各种各样的负载，因此，工作负载统计信息演变为除了cpuspeed以外，其他都是不恒定的。图7-3展示了我在一个生产系统上测量的工作负载统计信息的演变。为了生成这些图表，我以1小时为间隔、持续4天来收集工作负载统计信息。为此所写的SQL语句的例子请参阅system_stats_history.sql和system_stats_history_job.sql脚本。

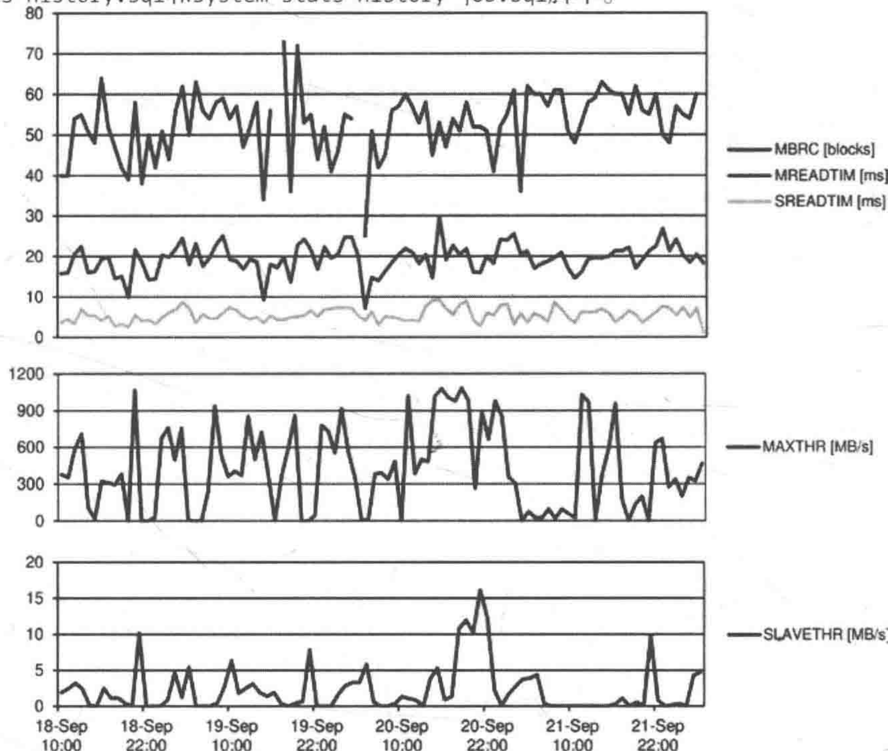


图7-3 在大多数系统上，工作负载统计信息的演变表现为除了恒定之外的各种形式

要避免在无法提供有代表性负载的时间段内收集工作负载统计信息，在我看来只有两种途径。要么通过持续几天的时间来收集工作负载统计信息，要么基于更短的时间段（例如，10分钟）来生成图表，以获取合理的值（如图7-3）。我通常建议使用后面的方法，因为当工作负载在同一时间段内变换非常频繁的时候，通过几天时间计算出来的结果可能会非常具有误导性。另外，使用更短的时间间隔，你也可以在相同的时间获取有用的系统性能的视图。

使用基于较短间隔的方法收集信息的另一个好处是，它迫使你不立刻更改数据字典中的系统统计信息。事实上，当收集系统统计信息时，更好的做法是把它们收集在一张备份表中并检查其一致性。然后，如果这些统计信息没问题，再将它们导入到数据字典中。

举例来讲，根据图7-3，我建议为mbrc、mreadtim和sreadtim使用平均值，为maxthr和slavethr使用最大值。类似下面的PL/SQL代码块可能会用于手工设置工作负载统计信息。注意在使用set_system_stats存储过程设置工作负载统计信息之前，会通过使用delete_system_stats存储过程删除掉旧的系统统计信息：

```
BEGIN
  dbms_stats.delete_system_stats();
  dbms_stats.set_system_stats(pname => 'CPUSPEED', pvalue => 772);
  dbms_stats.set_system_stats(pname => 'SREADTIM', pvalue => 5.5);
  dbms_stats.set_system_stats(pname => 'MREADTIM', pvalue => 19.4);
  dbms_stats.set_system_stats(pname => 'MBRC',     pvalue => 53);
  dbms_stats.set_system_stats(pname => 'MAXTHR',   pvalue => 1136136192);
  dbms_stats.set_system_stats(pname => 'SLAVETHR', pvalue => 16870400);
END;
```

当出现一天或一周中的不同时段需要不同的工作负载统计信息集合的情况时，手工设置系统统计信息的方法也适用。不过，必须指出，我从没遇到过需要一组以上工作负载统计信息的情形。

7.3.3 在无工作负载统计信息和工作负载统计信息之间进行选择

在两种可用的系统统计信息之间进行选择其实是在简单性和可控性之间进行选择。如果简单性是问题关键，你或许会选择无工作负载统计信息。这是因为，正如在之前的章节中所描述的，无工作负载统计信息更加容易收集。

注意 最简单的方法是使用默认统计信息，你可以通过调用delete_system_stats来实现。对于某些数据库，这些默认的统计信息可能就是你所需要的全部。

然而，通过选择使用无工作负载统计信息的简单方法，你失去了对以下两个具体特性的控制。

- 当使用无工作负载统计信息时，初始化参数db_file_multiblock_read_count的值可能会影响由查询优化器执行的估算。第9章中会讲到，这是不理想的。而在工作负载统计信息中，这个参数的角色被统计信息mbrc取代。
- 只有使用工作负载统计信息，凭借maxthr和slavethr统计信息，你才可以控制并行操作的成本。

这两个特性只有在使用工作负载统计信息时才可用。基于这个原因，我认为工作负载统计信息优先级更高，并且通常会推荐它们。你收集工作负载统计信息时的额外投入会在长期内获得回报。

7.4 还原系统统计信息

每当通过dbms_stats包更改了系统统计信息，都会将当前的统计信息保存到另一张数据字典表（wri$_optstat_aux_history），而非简单地使用新的统计信息覆盖旧的，这张表保留着所有在保留期内出现的变化。其用途是，万一新的统计信息导致低效率的执行计划，能够还原旧的统计信息。

出于还原旧统计信息的目的，dbms_stats包提供了restore_system_stats存储过程。这个存储过程只接受一个单独的参数：用于指定目标时间的一个timestamp类型的值。统计信息被还原为在指定时间点使用的那些值。例如，下面的PL/SQL代码块会将系统统计信息还原为一天以前的样子：

```
BEGIN
  dbms_stats.delete_system_stats();
  dbms_stats.restore_system_stats(as_of_timestamp => systimestamp - INTERVAL '1' DAY);
END;
```

警告 为确保能够精确地还原指定的时间点所使用的系统统计信息，你必须在还原之前删除当前的系统统计信息。否则，还原的统计信息实际上是与当前所使用的所有统计信息合并的结果。

系统统计信息（对象统计信息也一样，因为它们是由相同的基础功能维护的）在历史表中保存一段由保留期指定的时间间隔。默认值是31天。你可以通过调用dbms_stats包中的get_stats_history_retention函数来显示当前值，如下所示：

```
SELECT dbms_stats.get_stats_history_retention() AS retention FROM dual
```

为了修改此保留期，dbms_stats包提供了alter_stats_history_retention存储过程。下面是一个调用将保留期设置为14天的例子：

```
dbms_stats.alter_stats_history_retention(retention => 14)
```

注意，使用alter_stats_history_retention存储过程时，下面的值具有特殊含义：

- NULL设置保留期为默认值；
- 0禁用历史记录；
- –1禁用历史记录的清除。

将statistics_level初始化参数设置为typical（即默认值）或all时，会自动清除比保留期指定的时间更旧的统计信息。一旦有必要进行手工清除时，dbms_stats提供了purge_stats存储过程。下面的调用清除了历史表中所有超过14天的统计信息：

```
dbms_stats.purge_stats(before_timestamp => systimestamp - INTERVAL '14' DAY)
```

要执行alter_stats_history_retention和purge_stats存储过程，你需要analyze any和analyze any dictionary系统权限。

7.5 使用备份表

用来管理系统统计信息的大部分dbms_stats存储过程都能够使用数据字典或者备份表进行工作。但是有一个存储过程只能使用数据字典进行工作：restore_system_stats存储过程。

尽管默认情况下所有操作都是针对数据字典执行的，但是如果您想要改而使用备份表，那么支持备份表的存储过程会为你提供三个参数。这三个参数如下所示。

- stattab指定数据字典之外的一张表的名称用于存储统计信息。默认值是NULL。
- statown指定由stattab参数指定的表所有者。默认值是NULL，此时使用的值是当前用户。
- statid是一个可选的标识符，用来识别存储在备份表中的多组统计信息，即由stattab和statown参数指定的那些。只有当Oracle identifier[①]受支持时才可用。

举例来说，下面的调用收集无负载统计信息并将其存储在名为mystats的备份表中，这张表的所有者是system用户：

```
dbms_stats.gather_system_stats(gathering_mode => 'noworkload',
                               statown        => 'system',
                               stattab        => 'mystats')
```

① 参考Oracle官方文档*SQL Language Reference*中关于identifier的定义。

要创建一张备份表，可以借助dbms_stats包中的create_stat_table存储过程。创建的关键是指定备份表的所有者（通过ownname参数）和表名（通过stattab参数）。此外，可选的tblspace参数指定将表创建在哪个表空间中。如果没有指定tblspace参数，则最终会在用户的默认表空间中创建表。下面是一个例子：

```
dbms_stats.create_stat_table(ownname  => user,
                             stattab  => 'mystats',
                             tblspace => 'users')
```

dbms_stats包提供了drop_stat_table存储过程来删除一张备份表。也可以使用正常的DROP TABLE语句来删除备份表。例如：

```
dbms_stats.drop_stat_table(ownname => user,
                           stattab => 'mystats')
```

7.6 管理操作的日志记录

除了restore_system_stats之外，所有用于管理系统统计信息的dbms_stats过程，都会将一些关于它们的活动的信息记录到数据字典中。这一信息可以通过dba_optstat_operations视图具体化，并且从12.1版本开始也可以通过dba_optstat_operation_tasks视图查看。注意，在多租户环境下，cdb的视图同样可用。下面这段来自system_stats_logging.sql脚本输出的摘录展示了查询该视图的一个例子。通过查询该视图，可以发现执行了哪些操作，它们是什么时候开始的，以及花费了多长时间：

```
SQL> VARIABLE now VARCHAR2(14)

SQL> BEGIN
  2    SELECT to_char(sysdate,'YYYYMMDDHH24MISS') INTO :now FROM dual;
  3    dbms_stats.delete_system_stats();
  4    dbms_stats.gather_system_stats('noworkload');
  5  END;
  6  /

SQL> SELECT operation, start_time,
  2         (end_time-start_time) DAY(1) TO SECOND(0) AS duration
  3  FROM dba_optstat_operations
  4  WHERE start_time > to_date(:now,'YYYYMMDDHH24MISS')
  5  ORDER BY start_time;

OPERATION            START_TIME                        DURATION
-------------------- --------------------------------- -----------
delete_system_stats  25-SEP-13 16.59.47.679829 +02:00  +0 00:00:00
gather_system_stats  25-SEP-13 16.59.47.688208 +02:00  +0 00:00:02
```

另外，自12.1版本起，可以看见操作执行时使用的参数。下面的查询证明了这一点：

```
SQL> SELECT x.*
  2  FROM dba_optstat_operations o,
  3       XMLTable('/params/param'
  4                PASSING XMLType(notes)
  5                COLUMNS name VARCHAR2(20) PATH '@name',
  6                        value VARCHAR2(20) PATH '@val') x
```

```
  7  WHERE start_time > to_date(:now,'YYYYMMDDHH24MISS')
  8  AND operation = 'gather_system_stats';

NAME                 VALUE
-------------------- ----------
gathering_mode       noworkload
interval             60
statid
statown
stattab
```

在12.1版本中，也可以通过dbms_stats包的report_single_stats_operation函数提取出某一操作的细节。输出支持不同的格式（文本、HTML以及XML）。下面的查询演示了如何生成一个文本报告：

```
SQL> SELECT dbms_stats.report_single_stats_operation(opid         => id,
  2                                                  detail_level => 'all',
  3                                                  format       => 'text')
  4  FROM dba_optstat_operations
  5  WHERE operation = 'gather_system_stats'
  6  AND start_time > to_date(:now,'YYYYMMDDHH24MISS');

-------------------------------------------------------------------------------------------
| Operation | Operation           | Start Time         | End Time           | Additional Info |
| Id        |                     |                    |                    |                 |
-------------------------------------------------------------------------------------------
| 4928      | gather_system_stats | 25-SEP-13          | 25-SEP-13          | Parameters:     |
|           |                     | 16.28.35.528238    | 16.28.37.105673    | [gathering_mode:|
|           |                     | +02:00             | +02:00             | noworkload]     |
|           |                     |                    |                    | [interval: 60]  |
|           |                     |                    |                    | [statid: ]      |
|           |                     |                    |                    | [statown: ]     |
|           |                     |                    |                    | [stattab: ]     |
-------------------------------------------------------------------------------------------
```

同时要注意，日志信息会被与之前描述的统计信息历史相同的机制清除掉。因此两者具有相同的保留期。

7.7 对查询优化器的影响

系统统计信息对查询优化器估算的成本有直接影响。大部分统计信息只要可用就会被一直使用。然而，有些统计信息只有在查询优化器估算某些特别的执行计划时才会使用。具体来说，mbrc只有涉及多块读时才会被使用；maxthr和slavethr只有当SQL语句被认为会以并行方式执行时才会使用。

本节会举例说明一些用法。其他用法会在第9章中讨论查询优化器如何估算全表扫描的成本时介绍。

警告　本节提供的公式都没有被Oracle公开，但有一个例外。一些测试表明这些公式能够描述查询优化器是如何估算给定操作的成本的。不管怎样，都不能证明它们在所有的情形中都是精确或正确的。提供这些公式的目的是给你一个关于系统统计信息是如何影响查询优化器的思路。本章描述的系统统计信息仅从10.1版本开始才可用。如果将初始化参数optimizer_features_enable设置为9.2.0.8，查询优化器的行为并不总是与这里描述的一样。因为这样的配置根本不常见，因此不再提供该条件下不同行为的更多信息。关于optimizer_features_enable的信息请参考第9章。

当系统统计信息可用时，查询优化器计算两个成本：I/O和CPU。第9章将描述对于大部分重要的访问路径，I/O成本是如何计算的。关于CPU成本的计算只有很少的信息可供访问。尽管如此，我们仍能够推测，就CPU而言，查询优化器使其每个操作都关联一个成本。 例如，公式7-1是用来计算访问一个列的CPU成本的。

公式7-1　访问一个列的估算CPU成本依赖于这个列在表中的位置。这个公式给出了访问一行数据的成本。如果访问了多行，则CPU的成本会按比例增加。第16章详细介绍了CPU的成本为什么会与列的位置有关

$$cpu_cost = column_position \cdot 20$$

接下来的例子基于cpu_cost_column_access.sql脚本，进一步阐明了公式7-1。首先创建出一个拥有9个列的表，插入一行数据，然后通过EXPLAIN PLAN语句，将访问9个列各自的CPU成本分别显示出来。参见第10章中关于此SQL语句的详细信息。注意一开始有35 757的初始CPU成本用于访问表，然后接下来的每个列，CPU的成本都递增20。而在同一时刻，I/O的成本是恒定的。因为所有列都存储在相同的数据库块中，所以这是合理的，因此读取它们所需要的物理读的数量对于所有的查询都是一致的：

```
SQL> CREATE TABLE t (c1 NUMBER, c2 NUMBER, c3 NUMBER,
  2                  c4 NUMBER, c5 NUMBER, c6 NUMBER,
  3                  c7 NUMBER, c8 NUMBER, c9 NUMBER);

SQL> INSERT INTO t VALUES (1, 2, 3, 4, 5, 6, 7, 8, 9);

SQL> EXPLAIN PLAN SET STATEMENT_ID 'c1' FOR SELECT c1 FROM t;
SQL> EXPLAIN PLAN SET STATEMENT_ID 'c2' FOR SELECT c2 FROM t;
SQL> EXPLAIN PLAN SET STATEMENT_ID 'c3' FOR SELECT c3 FROM t;
SQL> EXPLAIN PLAN SET STATEMENT_ID 'c4' FOR SELECT c4 FROM t;
SQL> EXPLAIN PLAN SET STATEMENT_ID 'c5' FOR SELECT c5 FROM t;
SQL> EXPLAIN PLAN SET STATEMENT_ID 'c6' FOR SELECT c6 FROM t;
SQL> EXPLAIN PLAN SET STATEMENT_ID 'c7' FOR SELECT c7 FROM t;
SQL> EXPLAIN PLAN SET STATEMENT_ID 'c8' FOR SELECT c8 FROM t;
SQL> EXPLAIN PLAN SET STATEMENT_ID 'c9' FOR SELECT c9 FROM t;

SQL> SELECT statement_id, cpu_cost AS total_cpu_cost,
  2         cpu_cost-lag(cpu_cost) OVER (ORDER BY statement_id) AS cpu_cost_1_coll,
  3         io_cost
  4  FROM plan_table
  5  WHERE id = 0
  6  ORDER BY statement_id;
```

```
STATEMENT_ID TOTAL_CPU_COST CPU_COST_1_COLL IO_COST
------------ -------------- --------------- -------
c1                    35757                       3
c2                    35777              20       3
c3                    35797              20       3
c4                    35817              20       3
c5                    35837              20       3
c6                    35857              20       3
c7                    35877              20       3
c8                    35897              20       3
c9                    35917              20       3
```

I/O和CPU成本是按照不同的测量单位来表示的。很显然，一个SQL语句的总体成本不能简单地将这些成本累加计算。为了解决这个问题，查询优化器将使用引入了工作负载统计信息的公式7-2。简单来说，CPU成本除以cpuspeed来获取估计的消耗时间，然后除以sreadtim以使用与io_cost一样的测量单位来表示成本。

公式7-2 总体成本基于I/O成本和CPU成本

$$cost \approx io_cost + \frac{cpu_cost}{cpuspeed \cdot sreadtim \cdot 1000}$$

为了使用无负载统计信息计算整体成本，在公式7-2中cpuspeed被cpuspeednw替换，还有sreadtim的值由公式7-3计算出来。简单来讲，为计算sreadtim，公式7-3将定位磁盘上一个数据块所需的时间与将这个块传递至数据库引擎所需的时间相加。

公式7-3 如有必要，sreadtim会根据无负载统计信息和数据库默认的块大小进行计算

$$sreadtim \approx ioseektim + \frac{db_block_size}{iotfrspeed}$$

一般而言，如果工作负载统计信息是可用的，查询优化器就会使用它们而忽略无负载统计信息。应该清楚的是，查询优化器会执行一些健全性检查，这些检查可能禁用工作负载统计信息或者部分替换工作负载统计信息。可以通过脚本system_stats_sanity_checks.sql观察到这种行为。下面是一些观察的条目。

- 当mbrc不可用或者设置为0时，查询优化器会忽略工作负载统计信息，使用无负载统计信息。
- 当sreadtim不可用或者设置为0时，查询优化器会使用公式7-3和公式7-4分别重新计算sreadtim和mreadtim的值。
- 当mreadtim不可用时，或者当它没有sreadtim的值大时，查询优化器会使用公式7-3和公式7-4分别重新计算sreadtim和mreadtim的值。

公式7-4 mreadtim的计算基于无负载统计信息以及数据库的默认块大小

$$mreadtim \approx ioseektim + \frac{mbrc \cdot db_block_size}{iotfrspeed}$$

仅当在exadata模式下收集的无负载统计信息可用时，才会出现使用公式7-3和公式7-4的特例。事实上，伴随着这种类型的统计信息，所有的估算都是基于mbrc、ioseektim以及iotfrspeed的。

在被认为是并行执行的SQL语句的估算中，slavethr和maxthr又起到什么作用呢？简单而言，前

者可以增加并行执行的成本，后者可以通过高并行度降低并行执行的成本。接下来会详细讨论一下这两组统计信息的影响。

如果没有设置slavethr和maxthr，那么查询优化器会认为一个操作并行执行的成本与用于执行的并行度成反比，如公式7-5所示。因此，查询优化器认为无论并行度是多少，每个并行运行的从属进程都能够支撑公式7-6计算出来的吞吐率。

公式7-5　并行I/O成本与并行度成反比。注意，常量0.9是一个假想的因数，可能是考虑了并行处理过程中不可避免的竞争因素。

$$parallel_io_cost \approx \frac{serial_io_cost}{dop \cdot 0.9}$$

公式7-6　单个服务进程的预期吞吐率（以字节每秒为单位）的计算建立在工作负载统计信息和数据库默认块大小的基础上

$$mreadthr \approx \frac{mbrc \cdot db_block_size}{mreadtim} \cdot 1000$$

为了防止查询优化器在估算并行操作时过于乐观，可以通过slavethr增加估算的成本。要达到这个目的，可以将slavethr设置为一个低于mreadthr的值，后者的值是通过公式7-6计算出来的。换句话说，就是通知查询优化器每个从属进程的吞吐率要低于默认值。请注意，反之，通过给slavethr设置一个高于mreadthr的值来降低成本是不可能的。事实上，当slavethr和mreadthr的比例大于0.9（公式7-5使用的假想因数），则对查询优化器的成本估算没有影响。图7-4展示了对于一次全表扫描，将slavethr的值设置为mreadthr的一半时的影响。

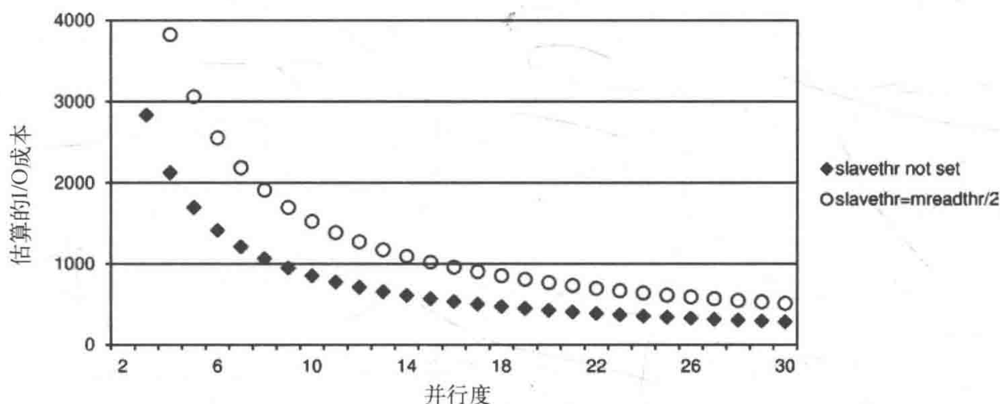


图7-4　使用slavethr和不使用slavethr估算的I/O成本对比（由parallel_fts_costing.sql脚本产生的数据）

应给予slavethr如公式7-7中所示的调整。注意它与公式7-5的区别：假想因数（0.9）只有在其大于slavethr和mreadthr的比例时才会被使用。

公式7-7　当slavethr和mreadthr的比例小于0.9时（关于k的定义参见该公式注解），发生的并行I/O成本的调整（增加）

$$parallel_io_cost \approx \frac{serial_io_cost}{dop \cdot \text{least}\left(0.9, \frac{slavethr \cdot k}{mreadthr}\right)}$$

注意 在公式7-7和公式7-8中，因数k依赖于数据库的版本。直到11.2.0.3版本为止，它的值都是1000。从11.2.0.4版本开始，它的值变成了1。因此，在11.2.0.3版本中，slavethr和mreadthr的比值仅对查询优化器估算的值有非常小的影响。基于这个原因，实际上在11.2.0.3版本中，大多数时候观察不到其影响。

如公式7-7明确显示的，成本和并行度成反比，因此slavethr只能用来增加成本，不能用于降低成本。实际上，真实的资源消耗并不总是与并行度成反比。事实上，因为数据库服务不会无限扩展，对于高并行度操作估算的成本太低。这恰恰是为什么maxthr可用的原因。图7-5展示了对于与图7-4所列举的案例同样的案例，设置maxthr对于预防成本降至某一特定界限之下的影响。注意尽管最低的成本与slavethr无关，降低成本在不同的并行度都有发生。

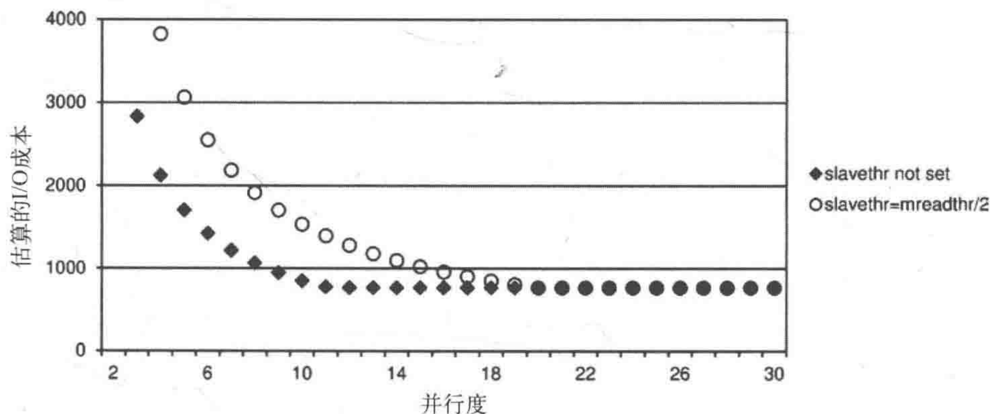


图7-5 设置maxthr的情况下估算的I/O成本对照（由parallel_fts_costing.sql脚本产生的数据）

如图7-5所示，maxthr的值随着并行度变得太高而停止降低成本。简单来说，查询优化器根据公式7-8计算，成本不能低于某一特定的值。

公式7-8 预期的单个服务进程吞吐率与整个系统的最大磁盘I/O吞吐率的比例限制并行I/O的成本（注意前面k的定义）

$$minimum_parallel_io_cost \approx serial_io_cost \cdot \frac{mreadthr}{maxthr \cdot k}$$

正如本节中所讨论的，系统统计信息使得查询优化器能够了解数据库引擎所运行的系统。这意味着，对于一个成功的配置，它们是基本要素。建议为了产生执行计划的稳定性而冻结它们。换言之，我把它们看成初始化参数。

当然，万一主要硬件或者软件发生了变化，系统统计信息就应该重新计算，因此应该检查整个配置情况。出于检查的目的，也应该定期把它们收集到备份表中（也就是说使用带有statown和stattab参数的gather_system_stats存储过程）并且验证当前值与数据字典中存储的值是否有重大差别。

7.8　小结

本章描述了什么是系统统计信息以及为何查询优化器需要它们。简单来说，它们提供关于CPU和磁盘I/O子系统的性能信息。本章还涉及如何使用dbms_stats包来管理系统统计信息，以及在数据字典中如何找到它们。

然而系统统计信息并不完全足以描述查询优化器运行的环境。查询优化器还需要深入了解存储在数据库中的数据。出于这个目的，可以使用另一种类型的统计信息：对象统计信息。下一章将提供该类型的统计信息的完整描述。

第 8 章

对象统计信息

8

对象统计信息描述在数据库中存储的数据。例如，它们告诉查询优化器表中存储了多少条数据。没有这些特定信息，查询优化器永远无法做出正确的决定，例如为小表或者大表（或结果集）找出正确的联接方法。为了说明这一点，可以参考下面的例子。比如我问你从一个地方到家最快的交通方式是什么。是乘坐汽车、火车还是飞机？为什么不骑自行车呢？问题的关键是，如果不考虑我的实际位置以及我的家在哪，你没法得到有效答案。没有对象统计信息，查询优化器也会存在相同的问题。它完全没法产生最优的执行计划。

本章首先介绍了哪些对象统计信息可供使用，以及在数据字典中如何找到它们。随之呈现的是dbms_stats包，该包用于收集、还原、锁定、对比和删除统计信息。最后，介绍一些用来管理对象统计信息的策略，充分利用可用的特性。至于查询优化器拿对象统计信息做什么，在这里只会进行简单介绍。大部分统计信息的用途将会在第9章中介绍。因为查询优化器会同时使用统计信息和初始化参数，所以在同一章中一起描述它们再合适不过了。

注意 通过ASSOCIATE STATISTICS语句，数据库引擎可以将用户定义的统计信息与列、函数、包、类型、应用域索引以及索引类型相关联。在需要时，这个SQL语句的功能非常强大，尽管在实践中这项技术很少会用到。基于这个原因，ASSOCIATE STATISTICS在这里就不多讲了。要查看相关信息，请参考*Oracle Database Data Cartridge Developer's Guide*手册以及*Expert Oracle Practices*（Apress，2010）的第7章。

8.1 dbms_stats 包

过去，对象统计信息是由ANALYZE语句收集的。现在已经不这样做了。对于收集对象统计信息，ANALYZE语句仍然可用，但只是用于向后兼容性的目的。自从Oracle9起，推荐使用dbms_stats包替代。实际上，dbms_stats包不仅提供更多的新特性，在某些情形下它还能提供更好的统计信息。举例来说，ANALYZE语句对统计信息收集提供的控制更少，不支持外部表，并且对于分区的对象，只会对每个segment分别收集统计信息，然后在表/索引级别提取出统计信息（通常少得可怜）。基于以上原因，本章将不涉及ANALYZE语句。

认识到这一点很重要：dbms_stats包提供一组全面的用于管理对象统计信息的存储过程和函数。

因为数据库中有许多对象，通过不同的粒度来管理它们的统计信息就显得非常重要。可以选择为整个数据库、数据字典、单个模式、单张表、单个索引或者是单独的表或索引分区来管理对象统计信息。

默认情况下，dbms_stats包直接修改数据字典中的数据。不过，它的许多存储过程和函数，也能使用存储在数据字典之外的用户定义的表进行工作。我将其称为备份表。

因为管理统计信息比单纯收集统计信息更为复杂，dbms_stats包提供以下关键特性（见图8-1）。

- 收集统计信息，并且可以选择在覆盖当前统计信息之前将它们存储到一张备份表中。
- 锁定和解锁存储在数据字典中的对象统计信息。
- 将对象统计信息从一个分区或子分区复制到另外一个分区或子分区。
- 还原数据字典中的对象统计信息。
- 删除存储在数据字典或备份表中的对象统计信息。
- 将对象统计信息从数据字典导出到备份表中。
- 将对象统计信息从备份表导入到数据字典中。
- 获取（提取）存储在数据字典或备份表中的对象统计信息。
- 设置（修改）存储在数据字典或备份表中的对象统计信息。

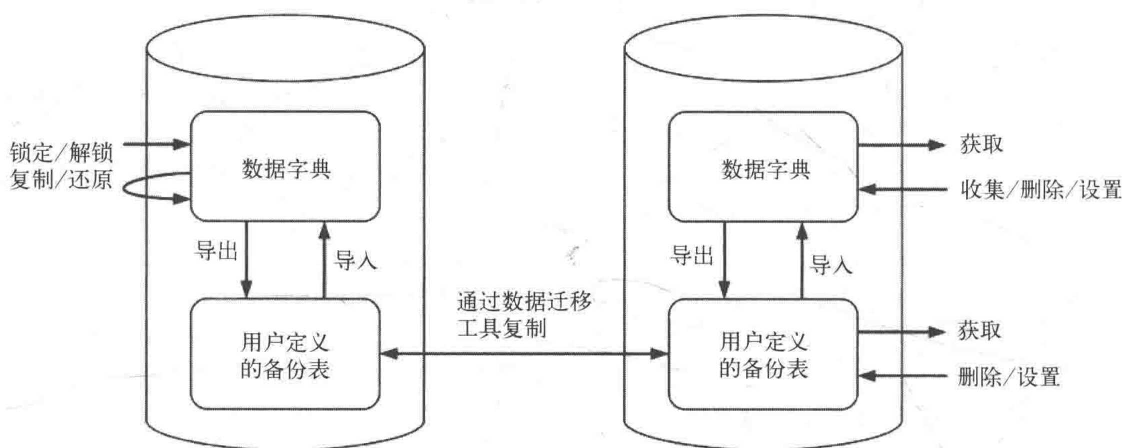


图8-1　dbms_stats包提供一组全面的用于管理对象统计信息的功能

注意，在数据库之间移动统计信息是借助通用的数据迁移工具（例如，数据泵）来完成的，并非是使用dbms_stats包本身。

随着粒度和执行操作的不同，表8-1列出了dbms_stats包提供的不同的存储过程和函数。举例来讲，如果想在一个单独的模式下执行，dbms_stats提供了gather_schema_stats、delete_schema_stats、lock_schema_stats、unlock_schema_stats、restore_schema_stats、export_schema_stats以及import_schema_stats。

表8-1　dbms_stats包提供的功能

特　性	数据库	数据字典	模　式	表*	索　引*
收集/删除	✓	✓	✓	✓	✓
锁定/解锁			✓	✓	

（续）

特 性	数据库	数据字典	模 式	表*	索 引*
复制				✓	
还原	✓	✓	✓	✓	
导出/导入	✓	✓	✓	✓	✓
获取/设置				✓	✓

* 对于分区对象，将处理限制到单个分区是可能的。

8.2 有哪些对象统计信息可用

对象统计信息分为三种类型：表统计信息、列统计信息以及索引统计信息。对于每种类型，又分为多达三种子类型：表/索引级别统计信息、分区级别统计信息以及子分区级别统计信息。显而易见，分区和子分区统计信息只有当对象分别进行了分区和划分了子分区时才可用。

对象统计信息通过表8-2中列举的数据字典视图来显示。当然，对于每一个视图都有dba、all，在12.1多租户环境下还有cdb版本可用，例如dba_tab_statistics、all_tab_statistics和cdb_tab_statistics。

表8-2 显示对象统计信息的数据字典视图关系表

对 象	表/索引级别统计信息	分区级别统计信息	子分区级别统计信息
表	user_tab_statistics	user_tab_statistics	user_tab_statistics
列	user_tab_col_statistics	user_part_col_statistics	user_subpart_col_statistics
	user_tab_histograms	user_part_histograms	user_subpart_histograms
索引	user_ind_statistics	user_ind_statistics	user_ind_statistics

本节的剩余部分描述在数据字典中可访问的最重要的对象统计信息。出于这一目的，我使用下面的SQL语句创建了一张测试表。这些SQL语句，与本节中所有其他的查询一样，都可以在脚本object_statistics.sql中找到：

```
CREATE TABLE t
AS
SELECT rownum AS id,
       50+round(dbms_random.normal*4) AS val1,
       100+round(ln(rownum/3.25+2)) AS val2,
       100+round(ln(rownum/3.25+2)) AS val3,
       dbms_random.string('p',250) AS pad
FROM dual
CONNECT BY level <= 1000
ORDER BY dbms_random.value;

UPDATE t SET val1 = NULL WHERE val1 < 0;

ALTER TABLE t ADD CONSTRAINT t_pk PRIMARY KEY (id);

CREATE INDEX t_val1_i ON t (val1);
```

```
CREATE INDEX t_val2_i ON t (val2);

BEGIN
  dbms_stats.gather_table_stats(
    ownname          => user,
    tabname          => 'T',
    estimate_percent => 100,
    method_opt       => 'for columns size skewonly id, val1 size 15, val2, val3 size 5, pad',
    cascade          => TRUE
  );
END;
/
```

8.2.1　表统计信息

接下来的查询展示了如何获取对于一张表来说最重要的表统计信息：

```
SQL> SELECT num_rows, blocks, empty_blocks, avg_space, chain_cnt, avg_row_len
  2  FROM user_tab_statistics
  3  WHERE table_name = 'T';

NUM_ROWS BLOCKS EMPTY_BLOCKS AVG_SPACE CHAIN_CNT AVG_ROW_LEN
-------- ------ ------------ --------- --------- -----------
    1000     44            0         0         0         266
```

下面是对于查询返回的表统计信息的说明。

- num_rows是表中的数据行数量。
- blocks是表中高水位线以下数据块的数量。
- empty_blocks是表中高水位线以上数据块的数量。dbms_stats包不会将这个值计算在内。这个值会被设置为0（除非有另外一个值已经存在于数据字典中）。
- avg_space是表的数据块中的平均空闲空间（按字节表示）。dbms_stats包不会将这个值计算在内。这个值会被设置为0（除非有另外一个值已经存在于数据字典中）。
- chain_cnt是表中链接和迁移到另一个块的数据行的总数（详见第16章）。即使查询优化器使用这个值，dbms_stats包也不会将其计算在内。它会被设置为0（除非有另外一个值已经存在于数据字典中）。
- avg_row_len是表中数据行的平均大小（按字节表示）。

高水位线

高水位线是段（segment）中已使用空间和未使用空间的分界线。已使用的块位于高水位线以下，未使用的块位于高水位线以上。高水位线以上的块从未被使用过或者初始化过。

通常情况下，请求空间的操作（例如，INSERT语句）只有当高水位线以下没有更多的空闲空间时才会提高高水位线。这里有一个常见的例外是在直接路径插入期间，因为它们专门使用高水位线以上的块（参考第15章）。

释放空间的操作（例如DELETE语句）并不会降低高水位线。它们只是使空间对其他操作可用。

如果释放空闲空间的速率等于或低于重用空间的速率，那么使用高水位线以下的数据块应该是最理想的。否则，高水位线以下的空闲空间会稳步增长。从长远来看，这样不仅会造成段大小的不必要增大，同时也会导致性能不理想。实际上，全表扫描会访问高水位线以下的所有块。即使这些块是空的也会扫描。应该通过重构段来解决这个问题。

8.2.2 列统计信息

下面的查询展示了如何获得对于一张表来说最重要的列统计信息：

```
SQL> SELECT column_name AS "NAME",
  2         num_distinct AS "#DST",
  3         low_value,
  4         high_value,
  5         density AS "DENS",
  6         num_nulls AS "#NULL",
  7         avg_col_len AS "AVGLEN",
  8         histogram,
  9         num_buckets AS "#BKT"
 10  FROM user_tab_col_statistics
 11  WHERE table_name = 'T';

NAME  #DST LOW_VALUE          HIGH_VALUE            DENS #NULL AVGLEN HISTOGRAM         #BKT
---- ----- ------------------ ------------------ ------- ----- ------ --------------- -----
ID    1000 C102               C20B                .00100     0      4 NONE                1
VAL1    22 C128               C140                .03884     0      3 HYBRID             15
VAL2     6 C20202             C20207              .00050     0      4 FREQUENCY           6
VAL3     6 C20202             C20207              .00050     0      4 TOP-FREQUENCY       5
PAD   1000 202623436F294373342 7E79514A202D4946493 .00100    0    251 HYBRID            254
           37B426574336E4A5B30 66C744E253F36264C69
           2E4F4B53236932303A2 27557A57737C6D4B225
           1215F462B7667457032 9414C442D2544364130
           694174782F7749393B6 612F5B3447405A4E714
           5735646366D20736939 A403B6237592B3D7B67
           335D712B233B3F      7D4D594E766B57
```

下面是对这个查询返回的列统计信息的说明。

- num_distinct是这个列非重复值的数量。
- low_value是这个列的最小值。它是通过内部形式显示的。注意，对于字符串列（在本例中是pad列），只有前32个字节（在12.1版本中是前64个字节）会被使用。
- high_value是这个列的最大值。它是通过内部形式显示的。注意，对于字符串列（在本例中是pad列），只有前32个字节（在12.1版本中是前64个字节）会被使用。

LOW_VALUE和HIGH_VALUE格式

遗憾的是，列low_value和high_value并不容易去判读。实际上，它们使用数据库引擎存储数据所使用的二进制内部形式来显示。要将它们转换为可读的值，有两种方式可行。

第一种方式，使用utl_raw包提供的函数cast_to_binary_double、cast_to_binary_float、

cast_to_binary_integer、cast_to_number、cast_to_nvarchar2、cast_to_raw，以及cast_to_varchar2。正如这些函数的名称所暗示的，对于每一种数据类型，都有对应的函数用于将内部值转化为实际值。比如，要获取val1列的最小值和最大值，可以使用以下查询：

```
SQL> SELECT utl_raw.cast_to_number(low_value) AS low_value,
  2         utl_raw.cast_to_number(high_value) AS high_value
  3  FROM user_tab_col_statistics
  4  WHERE table_name = 'T'
  5  AND column_name = 'VAL1';

LOW_VALUE HIGH_VALUE
--------- ----------
       39         63
```

第二种方式，使用dbms_stats包提供的存储过程convert_raw_value（重载了几次）、convert_raw_value_nvarchar以及convert_raw_value_rowid。注意，为避免使用PL/SQL代码块，下面的查询使用了版本12.1，使用该版本有可能在WITH子句中声明PL/SQL函数和过程。这个查询的用途与之前的查询（在脚本object_statistics.sql中，可以找到这个查询的变体支持的所有最常见的数据类型）是一样的：

```
SQL> WITH
  2    FUNCTION convert_raw_value(p_value IN RAW) RETURN NUMBER IS
  3      l_ret NUMBER;
  4    BEGIN
  5     dbms_stats.convert_raw_value(p_value, l_ret);
  6     RETURN l_ret;
  7    END;
  8  SELECT convert_raw_value(low_value) AS low_value,
  9         convert_raw_value(high_value) AS high_value
 10  FROM user_tab_col_statistics
 11  WHERE table_name = 'T'
 12  AND column_name = 'VAL1'
 13  /

LOW_VALUE HIGH_VALUE
--------- ----------
       39         63
```

- density是一个0到1之间的小数。值接近0表示在这个列上的限制条件会过滤掉大部分记录。值接近1表示在这个列上的限制条件几乎不会过滤掉任何记录。如果没有出现直方图，则density值为1/num_distinct。如果有直方图出现，则其值的计算方式会不同，且依赖于直方图的类型。不管怎样，从10.2.0.4版本起，对于有直方图的列，这个值仅用于在optimizer_features_enable初始化参数设置为更旧的版本时保持向后兼容性。
- num_nulls是存储在这个列上的NULL值的数量。
- avg_col_len是以字节表示的平均列大小。
- histogram表明这个列上是否有直方图可供使用，以及当直方图可用时直方图的类型是什么。

有效的值包括NONE（即没有直方图）、FREQUENCY、HEIGHT BALANCED，还有自12.1版本开始可用的TOP-FREQUENCY和HYBRID。

- num_buckets是直方图中桶的数量。一个桶，或者在统计信息里称之为类别（category），是具有相同类型的一组值。正如在下一节中所描述的，直方图由至少一个桶组成。如果没有直方图，这个值设置为1。到11.2版本为止，桶的最大数量为254，而从12.1版本开始起，桶的最大数量为2048。

8.2.3 直方图

查询优化器以“数据是均匀分布的”这一原则为出发点。贯穿前面部分的测试表存储在ID列上的数据正是一个均匀分布的数据集的例子。实际上，它将1~1000之间的每个整数正好存储一次。在这种情形下，要生成根据该列上的谓词条件（例如id BETWEEN 6 AND 19）过滤掉的记录数的合理估算，查询优化器仅需要描述谓词部分的对象统计信息：最小值、最大值以及非重复值的数量。

如果数据并非均匀分布的，那么查询优化器在没有额外信息的情况下就无法计算合理的估算。举例来说，对于存储在val2列上的已知数据集合（见下面查询的输出部分），查询优化器如何对像val2=105这样的谓词做出有意义的估算？答案是不能，因为查询优化器丝毫不知道有大概50%的记录满足这个谓词条件：

```
SQL> SELECT val2, count(*)
  2  FROM t
  3  GROUP BY val2
  4  ORDER BY val2;

      VAL2   COUNT(*)
---------- ----------
       101          8
       102         25
       103         68
       104        185
       105        502
       106        212
```

查询优化器需要的关于非均匀分布数据的额外信息被称作*直方图*（histogram）。在12.1版本之前，有两种类型的直方图可用：*频率直方图*（frequency histogram）和*高度均衡直方图*（height-balanced histogram）。Oracle Database 12.1引入了两种额外的直方图来取代高度均衡直方图：*高频率直方图*（top frequency histogram）和*混合直方图*（hybrid histogram）。

警告 只有当收集对象统计信息时使用dbms_stats.auto_sample_size（在本章稍后的8.3.2节会介绍这个主题）作为采样频率时，dbms_stats包才会创建高频率直方图和混合直方图。

---

1. 频率直方图

频率直方图就是大多数人对直方图这一概念的理解。图8-2是频率直方图的一个例子，也是对之前查询返回数据的一个直观图示。

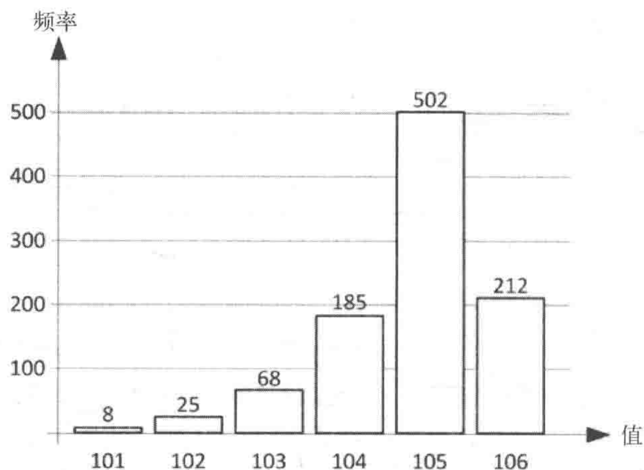


图8-2 根据存储在val2列中的数据集绘制的频率直方图的图示

存储在数据字典中的频率直方图与这个图示很像。主要的区别是字典中不是使用频率，而是使用累积频率。下面的查询通过计算两个相邻桶的值（注意，endpoint_number即是累计频率）之间的差来将累计频率转化为频率：

```
SQL> SELECT endpoint_value, endpoint_number,
  2         endpoint_number - lag(endpoint_number,1,0)
  3                           OVER (ORDER BY endpoint_number) AS frequency
  4  FROM user_tab_histograms
  5  WHERE table_name = 'T'
  6  AND column_name = 'VAL2'
  7  ORDER BY endpoint_number;

ENDPOINT_VALUE ENDPOINT_NUMBER FREQUENCY
-------------- --------------- ---------
           101               8         8
           102              33        25
           103             101        68
           104             286       185
           105             788       502
           106            1000       212
```

频率直方图的核心特性如下所述。

- 桶的数量（换句话说，类别的数量）与不重复值的数量一致。在像user_tab_histograms这样的视图中每个桶都有一条对应的记录可用。
- endpoint_value列提供其自身值的一个数值形式表示。因此，对于非数值形式的数据类型，必须将其实际值编码为一个数字。根据数据、数据类型以及版本的不同，实际值可能在endpoint_actual_value列（在之前的输出中并没有显示）中可见。要非常清楚地了解存储在直方图中的值，只能根据前面32个字节（在12.1版本中是64个字节）来区分。结果就是拥有较长固定前缀的值可能会危及直方图的有效性。尤其是当使用每个字符可能占用四个字节的多字节字符集时更是如此。

- endpoint_number列提供值的累积频率。要获得真正的频率，必须减去前一条记录的endpoint_number列的值。

警告 假如动态采样用于构建直方图，则频率信息应根据采样大小按比例决定。要知道比例因子，请用采样大小（sample_size）除以记录的数量（num_rows）。这两个列都是由类似user_tab_statistics这样的视图提供的。

接下来的例子展示了查询优化器如何利用频率直方图来精确估算一个在val2列（有关EXPLAIN PLAN语句的详细信息，请参见第10章）上使用了谓词的查询返回的记录数量（cardinality）：

```
SQL> EXPLAIN PLAN SET STATEMENT_ID '101' FOR SELECT * FROM t WHERE val2 = 101;
SQL> EXPLAIN PLAN SET STATEMENT_ID '102' FOR SELECT * FROM t WHERE val2 = 102;
SQL> EXPLAIN PLAN SET STATEMENT_ID '103' FOR SELECT * FROM t WHERE val2 = 103;
SQL> EXPLAIN PLAN SET STATEMENT_ID '104' FOR SELECT * FROM t WHERE val2 = 104;
SQL> EXPLAIN PLAN SET STATEMENT_ID '105' FOR SELECT * FROM t WHERE val2 = 105;
SQL> EXPLAIN PLAN SET STATEMENT_ID '106' FOR SELECT * FROM t WHERE val2 = 106;

SQL> SELECT statement_id, cardinality
  2  FROM plan_table
  3  WHERE id = 0;

STATEMENT_ID CARDINALITY
------------ -----------
101                    8
102                   25
103                   68
104                  185
105                  502
106                  212
```

在上面的例子中，所有的谓词仅引用了直方图中体现的值。当使用其他值时，又会发生什么呢？直到10.2.0.3版本（包括10.2.0.3版本在内）为止，查询优化器都使用1作为频率。从10.2.0.4版本起开始，有两种不同的情况需要考虑。第一，如果使用的值在最小值和最大值之间，查询优化器取直方图中体现的所有值中最低的频率并将其除以2。第二，如果使用的值超出了直方图的涵盖范围，则频率依赖于到最小值或最大值的距离。下面的例子证明了这一点：

```
SQL> EXPLAIN PLAN SET STATEMENT_ID '096' FOR SELECT * FROM t WHERE val2 = 96;
SQL> EXPLAIN PLAN SET STATEMENT_ID '098' FOR SELECT * FROM t WHERE val2 = 98;
SQL> EXPLAIN PLAN SET STATEMENT_ID '100' FOR SELECT * FROM t WHERE val2 = 100;
SQL> EXPLAIN PLAN SET STATEMENT_ID '103.5' FOR SELECT * FROM t WHERE val2 = 103.5;
SQL> EXPLAIN PLAN SET STATEMENT_ID '107' FOR SELECT * FROM t WHERE val2 = 107;
SQL> EXPLAIN PLAN SET STATEMENT_ID '109' FOR SELECT * FROM t WHERE val2 = 109;
SQL> EXPLAIN PLAN SET STATEMENT_ID '111' FOR SELECT * FROM t WHERE val2 = 111;

SQL> SELECT statement_id, cardinality
  2  FROM plan_table
  3  WHERE id = 0
  4  ORDER BY statement_id;
```

```
STATEMENT_ID CARDINALITY
------------ -----------
096                    1
098                    2
100                    3
103.5                  4
107                    3
109                    2
111                    1
```

2. 高度均衡直方图

当不重复值的数量大于允许的桶的最大数量（使用dbms_stats包时，会有一个硬性限制，甚至有可能会指定一个更低的值）时，你就无法使用频率直方图，因为每个桶只支持一个单独的值。此时就该高度均衡直方图施展身手了。

要创建一个高度均衡直方图，考虑一下接下来的过程。首先，创建出一个频率直方图。然后，如图8-3所示，频率直方图的值被堆积成一“堆”。最后，这个“堆”再被分成几个具有相同高度的桶。例如，在图8-3中，“堆”被分到了五个桶中。

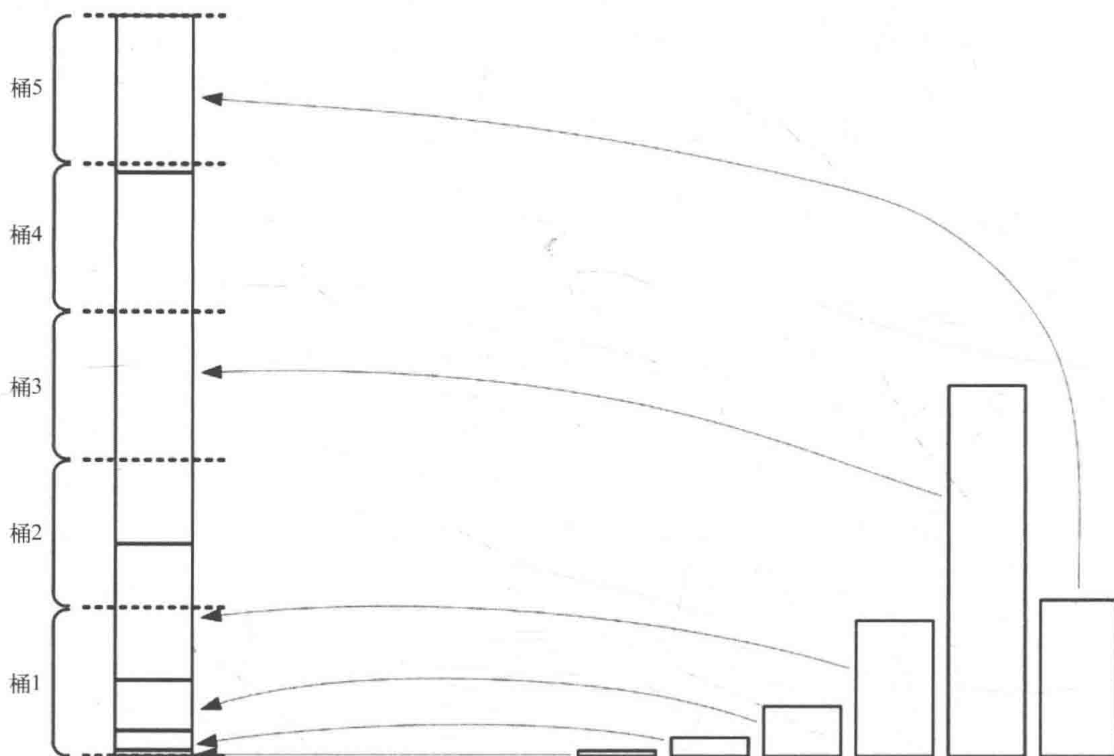


图8-3　将频率直方图转换为高度均衡直方图

下面的查询是一个如何为val2列生成一个高度均衡直方图的例子。图8-4展示了这个查询返回数据的一个图示。注意每个桶的端点值正是拆分数据出现的点。此外，桶0被添加进来用以存储最小值：

```
SQL> SELECT count(*), max(val2) AS endpoint_value, endpoint_number
  2  FROM (
  3    SELECT val2, ntile(5) OVER (ORDER BY val2) AS endpoint_number
  4    FROM t
  5  )
  6  GROUP BY endpoint_number
  7  ORDER BY endpoint_number;

  COUNT(*) ENDPOINT_VALUE ENDPOINT_NUMBER
---------- -------------- ---------------
       200            104               1
       200            105               2
       200            105               3
       200            106               4
       200            106               5
```

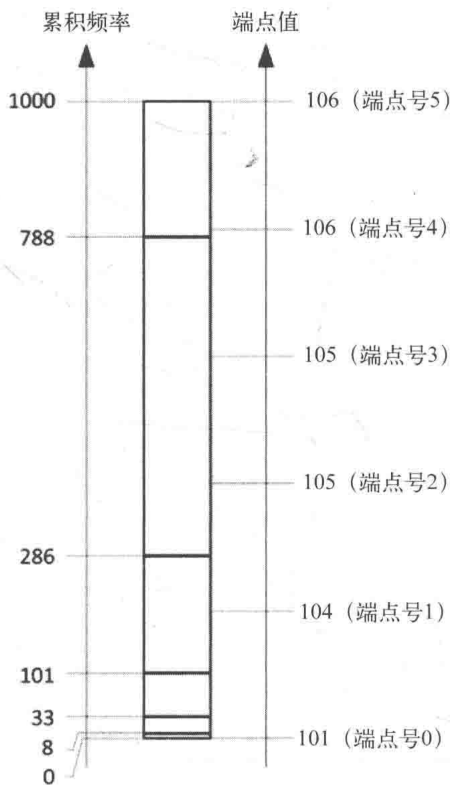


图8-4　根据存储在val2列上的数据集建立的高度均衡直方图

针对图8-4中的案例，接下来的查询展示了存储在数据字典中的高度均衡直方图。有趣的是，并没有存储所有的桶。之所以没有存储所有的桶是因为，几个拥有相同端点值的相邻桶没有多大用处。实际上，从显示出来的数据可以推断出，桶2的端点值是105，而桶4的端点值为106。查询结果有点浓缩的意思。在直方图中出现多次的值被称为常见值，并且会被查询优化器特殊处理：

```
SQL> SELECT endpoint_value, endpoint_number
  2  FROM user_tab_histograms
  3  WHERE table_name = 'T'
  4  AND column_name = 'VAL2'
  5  ORDER BY endpoint_number;

ENDPOINT_VALUE ENDPOINT_NUMBER
-------------- ---------------
           101               0
           104               1
           105               3
           106               5
```

下面是高度均衡直方图的主要特性。

- 桶的数量少于不重复值的数量。对于每一个桶，除非它们进行了压缩，否则都在像user_tab_histograms这样的视图中有一条带有端点号的记录与之对应。此外，端点号0表明是最小值。
- endpoint_value列给出关于值本身的数字表示形式。关于这个列的更多信息，请参考“频率直方图”一节中的描述。
- endpoint_number列给出桶的编号。
- 直方图不存储值的频率。

下面的例子展示了当存在合适的高度均衡直方图时查询优化器所作的估算。注意与频率直方图相比相对较低的精确度：

```
SQL> EXPLAIN PLAN SET STATEMENT_ID '101' FOR SELECT * FROM t WHERE val2 = 101;
SQL> EXPLAIN PLAN SET STATEMENT_ID '102' FOR SELECT * FROM t WHERE val2 = 102;
SQL> EXPLAIN PLAN SET STATEMENT_ID '103' FOR SELECT * FROM t WHERE val2 = 103;
SQL> EXPLAIN PLAN SET STATEMENT_ID '104' FOR SELECT * FROM t WHERE val2 = 104;
SQL> EXPLAIN PLAN SET STATEMENT_ID '105' FOR SELECT * FROM t WHERE val2 = 105;
SQL> EXPLAIN PLAN SET STATEMENT_ID '106' FOR SELECT * FROM t WHERE val2 = 106;

SQL> SELECT statement_id, cardinality
  2  FROM plan_table
  3  WHERE id = 0;

STATEMENT_ID CARDINALITY
------------ -----------
101                   50
102                   50
103                   50
104                   50
105                  400
106                  300
```

注意　你可能认为关于值105和106的基数估算完全一样（400，因为这两个高频率值都占了桶数量的2/5）。但是对于值106，却不是这样。这是因为，查询优化器在出现一个常见值同时也是直方图的最大值时，调整了估算的结果。

同样对于这种类型的直方图，我们来看一下当使用了直方图中没有体现的值时会发生什么。此时

需要考虑两种完全不同的情况。第一，如果值在最小值和最大值之间，查询优化器会使用与其他非常见值一样的频率。第二，如果值在直方图涵盖的值范围之外，则频率依赖于其到最小值或最大值的距离。下面的例子证明了这一点：

```
SQL> EXPLAIN PLAN SET STATEMENT_ID '096' FOR SELECT * FROM t WHERE val2 = 96;
SQL> EXPLAIN PLAN SET STATEMENT_ID '098' FOR SELECT * FROM t WHERE val2 = 98;
SQL> EXPLAIN PLAN SET STATEMENT_ID '100' FOR SELECT * FROM t WHERE val2 = 100;
SQL> EXPLAIN PLAN SET STATEMENT_ID '103.5' FOR SELECT * FROM t WHERE val2 = 103.5;
SQL> EXPLAIN PLAN SET STATEMENT_ID '107' FOR SELECT * FROM t WHERE val2 = 107;
SQL> EXPLAIN PLAN SET STATEMENT_ID '109' FOR SELECT * FROM t WHERE val2 = 109;
SQL> EXPLAIN PLAN SET STATEMENT_ID '111' FOR SELECT * FROM t WHERE val2 = 111;

SQL> SELECT statement_id, cardinality
  2  FROM plan_table
  3  WHERE id = 0
  4  ORDER BY statement_id;

STATEMENT_ID CARDINALITY
------------ -----------
096                    1
098                   20
100                   40
103.5                 50
107                   40
109                   20
111                    1
```

就这两种类型直方图的这些关键特性而论，很明显频率直方图要比高度均衡直方图更加精确。高度均衡直方图的主要问题不仅仅是精确度更低，而且有时候可能会意外导致一个值被当做常见值。例如，在图8-4所示的直方图中，桶4和桶5之间的拆分点非常接近于值从105变为106的点上。

因此，即使是数据分布非常微小的变化也可能导致一个不同的直方图以及不同的估算结果。在下面的例子中，只有20条记录被更新（约占总记录数的2%），就展示了这样的一种情况：

```
SQL> UPDATE t SET val2 = 105 WHERE val2 = 106 AND rownum <= 20;

SQL> REMARK at this point object statistics are gathered

SQL> SELECT endpoint_value, endpoint_number
  2  FROM user_tab_histograms
  3  WHERE table_name = 'T'
  4  AND column_name = 'VAL2'
  5  ORDER BY endpoint_number;

ENDPOINT_VALUE ENDPOINT_NUMBER
-------------- ---------------
           101               0
           104               1
           105               4
           106               5

SQL> EXPLAIN PLAN SET STATEMENT_ID '101' FOR SELECT * FROM t WHERE val2 = 101;
SQL> EXPLAIN PLAN SET STATEMENT_ID '102' FOR SELECT * FROM t WHERE val2 = 102;
SQL> EXPLAIN PLAN SET STATEMENT_ID '103' FOR SELECT * FROM t WHERE val2 = 103;
```

```
SQL> EXPLAIN PLAN SET STATEMENT_ID '104' FOR SELECT * FROM t WHERE val2 = 104;
SQL> EXPLAIN PLAN SET STATEMENT_ID '105' FOR SELECT * FROM t WHERE val2 = 105;
SQL> EXPLAIN PLAN SET STATEMENT_ID '106' FOR SELECT * FROM t WHERE val2 = 106;

SQL> SELECT statement_id, cardinality
  2  FROM plan_table
  3  WHERE id = 0;

STATEMENT_ID CARDINALITY
------------ -----------
101                   80
102                   80
103                   80
104                   80
105                  600
106                   80
```

因此，在实践中，高度均衡直方图可能不仅会令人误解，同时也会导致查询优化器估算的不稳定性。为了不再使用它们，自12.1版本开始，高频率直方图和混合直方图替代了高度均衡直方图。

3. 高频率直方图

频率直方图的一个关键特征是每个值都在直方图中体现出来。尽管这使得这些频率非常精确，但是因为桶的数量的限制，有时无法创建频率直方图。高频率直方图概念的真实意图，就是假使存在某些代表占比很小的数据的值，这些值可以被安全丢弃，因为它们在统计上无关紧要。而且如果能够丢弃足够多的值来避免超出桶的数量限制，那么就可能会创建出根据top-*n*值构造的高频率直方图。

为了确定使用*n*个桶的直方图是否足够精确，数据库引擎检查这*n*个值是否至少代表了百分比为*p*的数据量，而*p*是由公式8-1计算出来的。例如，类似val3列上建立的这个高频率直方图，它有5个桶，必须得代表至少80%（100−100/5）的数据量。

公式8-1　top-*n*值需要代表的数据量的最小百分比

$$p = 100 - \frac{100}{n}$$

在val3列的案例中，五个桶就足够了，因为从下面查询的输出来看，top-3的值已经占到行数据量的80%多：

```
SQL> SELECT val3, count(*) AS frequency, ratio_to_report(count(*)) OVER ()*100 AS percent
  2  FROM t
  3  GROUP BY val3
  4  ORDER BY val3;

      VAL3 FREQUENCY    PERCENT
---------- --------- ----------
       101         8        0.8
       102        25        2.5
       103        68        6.8
       104       185       18.5
       105       502       50.2
       106       212       21.2
```

接下来是存储在数据字典中的关于val3列的直方图：

```
SQL> SELECT endpoint_value, endpoint_number,
  2         endpoint_number - lag(endpoint_number,1,0)
  3                           OVER (ORDER BY endpoint_number) AS frequency
  4  FROM user_tab_histograms
  5  WHERE table_name = 'T'
  6  AND column_name = 'VAL3'
  7  ORDER BY endpoint_number;

ENDPOINT_VALUE ENDPOINT_NUMBER FREQUENCY
-------------- --------------- ---------
           101               1         1
           103              69        68
           104             254       185
           105             756       502
           106             968       212
```

对比val2列的频率直方图，有两点不同。首先，代表值102的桶并不存在。并且，这还是在值102的频率比值101的频率高的情况下。换句话说，这个直方图并不代表top-5的值。其次，代表值101的桶，尽管等于这个值的数据只有8条，但是其端点值endpoint_number却等于1。事实是任何一个直方图必须总是包含最小值和最大值。如果是像本例这样，两个值中有一个因为不是top-*n*值的一部分而被丢弃时，除最小值、最大值以外的值就被丢弃了（拥有最低频率的那一个），然后最小值/最大值的频率被设置为1。注意，在经过这样的操作之后，必须重新评估基于公式8-1的规则。

接下来的例子展示了这一点，正如你所期待的，查询优化器使用高频率直方图执行的估算与使用频率直方图进行的估算之间的区别仅在于没有频率信息的值（101和102）上：

```
SQL> EXPLAIN PLAN SET STATEMENT_ID '101' FOR SELECT * FROM t WHERE val3 = 101;
SQL> EXPLAIN PLAN SET STATEMENT_ID '102' FOR SELECT * FROM t WHERE val3 = 102;
SQL> EXPLAIN PLAN SET STATEMENT_ID '103' FOR SELECT * FROM t WHERE val3 = 103;
SQL> EXPLAIN PLAN SET STATEMENT_ID '104' FOR SELECT * FROM t WHERE val3 = 104;
SQL> EXPLAIN PLAN SET STATEMENT_ID '105' FOR SELECT * FROM t WHERE val3 = 105;
SQL> EXPLAIN PLAN SET STATEMENT_ID '106' FOR SELECT * FROM t WHERE val3 = 106;

SQL> SELECT statement_id, cardinality
  2  FROM plan_table
  3  WHERE id = 0
  4  ORDER BY statement_id;

STATEMENT_ID CARDINALITY
------------ -----------
101                   32
102                   32
103                   68
104                  185
105                  502
106                  212
```

注意，对于值101和102，其频率是直方图显示的所有值中最低的频率除以2得到的。注意，结果是32，可能并非你期待的34（68/2），因为并非所有的值都在直方图中体现了。

我们来看一下如果使用了直方图中没有体现的值会发生什么。简单来说，与频率直方图一样。下面的例子证实了这一点：

8

```
SQL> EXPLAIN PLAN SET STATEMENT_ID '096' FOR SELECT * FROM t WHERE val3 = 96;
SQL> EXPLAIN PLAN SET STATEMENT_ID '098' FOR SELECT * FROM t WHERE val3 = 98;
SQL> EXPLAIN PLAN SET STATEMENT_ID '100' FOR SELECT * FROM t WHERE val3 = 100;
SQL> EXPLAIN PLAN SET STATEMENT_ID '103.5' FOR SELECT * FROM t WHERE val3 = 103.5;
SQL> EXPLAIN PLAN SET STATEMENT_ID '107' FOR SELECT * FROM t WHERE val3 = 107;
SQL> EXPLAIN PLAN SET STATEMENT_ID '109' FOR SELECT * FROM t WHERE val3 = 109;
SQL> EXPLAIN PLAN SET STATEMENT_ID '111' FOR SELECT * FROM t WHERE val3 = 111;

SQL> SELECT statement_id, cardinality
  2  FROM plan_table
  3  WHERE id = 0
  4  ORDER BY statement_id;

STATEMENT_ID CARDINALITY
------------ -----------
096                    1
098                   13
100                   26
103.5                 32
107                   26
109                   13
111                    1
```

如果不满足基于公式8-1的规则，也就无法创建频率直方图和高频率直方图中的任何一种，数据库引擎会创建一个混合直方图。

4. 混合直方图

混合直方图综合了频率直方图和高度均衡直方图的一些特征。创建混合直方图与创建高度均衡直方图都是以同样的方法开始的。随后进行了以下两项重要的改进。

- 每个不重复值都关联到一个单独的桶（换句话说，为高度均衡直方图定义的常见值的概念不复存在）。基于该目的，桶的限制被转移了。结果，每个桶可能会根据不同数量的记录来创建。
- 频率被加入到每个桶的端点值中。因此，对于端点值，而且仅对于端点值而言，就有了某种频率直方图可用。

测试表有两个混合直方图。例如，我们看一下为val1列（注意，该列拥有22个不同的值）创建的那个混合直方图。下面查询的输出显示了该混合直方图包含的数据集：

```
SQL> SELECT val1, count(*), ratio_to_report(count(*)) OVER ()*100 AS percent
  2  FROM t
  3  GROUP BY val1
  4  ORDER BY val1;

      VAL1   COUNT(*) PERCENT
---------- ---------- -------
        39          2     0.2
        41          4     0.4
        42         13     1.3
        43         21     2.1
        44         26     2.6
        45         54     5.4
```

```
46         66     6.6
47         86     8.6
48         81     8.1
49         97     9.7
50        102    10.2
51        103    10.3
52         80     8.0
53         64     6.4
54         76     7.6
55         50     5.0
56         30     3.0
57         21     2.1
58         12     1.2
59          6     0.6
60          5     0.5
63          1     0.1
```

如果数据库引擎被要求创建有10个桶的直方图，那么不管是频率直方图还是高频率直方图，都无法应用。前者不适用是因为桶的数量小于不重复值的数量，后者是因为top-10的值仅代表约80%的记录（根据公式8-1需要代表90%的值；90 = 100–100/10）。因此，就有了提供以下信息的混合直方图：

```
SQL> SELECT endpoint_value, endpoint_number,
  2         endpoint_number - lag(endpoint_number,1,0)
  3                           OVER (ORDER BY endpoint_number) AS count,
  4         endpoint_repeat_count
  5  FROM user_tab_histograms
  6  WHERE table_name = 'T'
  7  AND column_name = 'VAL1'
  8  ORDER BY endpoint_number;

ENDPOINT_VALUE ENDPOINT_NUMBER      COUNT ENDPOINT_REPEAT_COUNT
-------------- --------------- ---------- ---------------------
            39               2          2                     2
            44              66         64                    26
            45             120         54                    54
            46             186         66                    66
            47             272         86                    86
            48             353         81                    81
            49             450         97                    97
            50             552        102                   102
            51             655        103                   103
            52             735         80                    80
            53             799         64                    64
            54             875         76                    76
            56             955         80                    30
            59             994         39                     6
            63            1000          6                     1
```

请注意，在上面的输出中，一方面，endpoint_number列提供关于每个桶上关联的记录数量的信息，而另一方面，endpoint_repeat_count列提供端点值的频率信息。根据这个信息，由查询优化器执行的关于端点值的估算就可以做到精确。下面是一个例子：

```
SQL> EXPLAIN PLAN SET STATEMENT_ID '44' FOR SELECT * FROM t WHERE val1 = 44;
SQL> EXPLAIN PLAN SET STATEMENT_ID '50' FOR SELECT * FROM t WHERE val1 = 50;
```

```
SQL> EXPLAIN PLAN SET STATEMENT_ID '56' FOR SELECT * FROM t WHERE val1 = 56;

SQL> SELECT statement_id, cardinality
  2  FROM plan_table
  3  WHERE id = 0
  4  ORDER BY statement_id;

STATEMENT_ID CARDINALITY
------------ -----------
44                    26
50                   102
56                    30
```

提示　混合直方图提供的信息要比高度均衡直方图提供的信息好得多。基于这个原因，从12.1版本开始，就可以并且也应该完全忽略掉高度均衡直方图了。

5. 无直方图

请注意，user_tab_histograms视图为每个没有直方图的列显示了两行数据，这非常有意义。这是因为最小值和最大值分别存储在端点号0和1上。举例来说，对于id列的内容，没有直方图可用，就会显示为下面的样子：

```
SQL> SELECT endpoint_value, endpoint_number
  2  FROM user_tab_histograms
  3  WHERE table_name = 'T'
  4  AND column_name = 'ID';

ENDPOINT_VALUE ENDPOINT_NUMBER
-------------- ---------------
             1               0
          1000               1
```

8.2.4　扩展统计信息

只有当谓词条件中使用了未经修改的列值时，上一节中描述的列统计信息和直方图才会起作用。例如，如果使用了谓词country='Switzerland'，通过country列上适当的列统计信息和直方图，查询优化器应该能够正确估算它的选择率。这是因为列统计信息和直方图描述的是country列自身的值。另一方面，如果使用了谓词upper(country)='SWITZERLAND'，查询优化器就不再能够直接从对象统计信息和直方图中推断出选择率了。当一个谓词条件引用了多个列时也会出现类似的问题。举个例子，如果将谓词条件country='Denmark' AND language='Danish'应用到一张包含全世界人口信息的表上，则很可能这两个限制条件都应用到了表中大多数记录的相同记录上了。实际上，大多数讲丹麦语的人生活在丹麦，生活在丹麦的大多数人讲丹麦语。换句话说，这两个限制条件几乎是冗余的。这样的列通常称作关联列（correlated column），并且它们会对查询优化器造成挑战。这是因为没有任何对象统计信息或者直方图描述这样互相依赖的数据，或者换句话说，查询优化器实际上是假设存储在不同列中的数据没有相互依赖关系。

自11.1版本开始，就可以做到基于表达式或者一组列来收集对象统计信息和直方图来解决这样的问题。这些新的统计信息称作扩展统计信息。这背后其实主要就是根据一个表达式或者一组列来创建一个叫作扩展信息的隐藏列。然后就在这个隐藏列上收集对象统计信息和直方图。

这个概念通过dbms_stats包的create_extended_stats函数来实现。例如，通过接下来的查询创建两个表达式。第一个是在upper(pad)上，第二个是由列val2和val3组成的一个列组。在测试表中，这些列包含完全一样的值；换句话说，这些列是高度关联的（实际上是完全关联的）。根据定义，如下面要展示的，表达式或这组列必须包含在一对圆括号中。注意，这个函数返回的是由系统生成的扩展信息名称（一个由SYS_STU开头的30个字节的名称）：

```
SQL> SELECT dbms_stats.create_extended_stats(ownname => user,
  2                                          tabname => 'T',
  3                                          extension => '(upper(pad))') AS ext1,
  4         dbms_stats.create_extended_stats(ownname => user,
  5                                          tabname => 'T',
  6                                          extension => '(val2,val3)') AS ext2
  7  FROM dual;

EXT1                           EXT2
------------------------------ ------------------------------
SYS_STU0KSQX64#I01CKJ5FPGFK3W9 SYS_STUPS77EFBJCOTDFMHM8CHP7Q1
```

注意　生成扩展信息的这组列不能引用表达式或虚拟列。

显然，一旦扩展信息[1]创建完毕，数据字典就可以提供关于它们的信息。下面的查询基于user_stat_extensions视图，显示了已经存在的测试表的扩展信息。视图同时还有dba、all以及在12.1多租户环境下的cdb版本：

```
SQL> SELECT extension_name, extension
  2  FROM user_stat_extensions
  3  WHERE table_name = 'T';

EXTENSION_NAME                 EXTENSION
------------------------------ ---------------
SYS_STU0KSQX64#I01CKJ5FPGFK3W9 (UPPER("PAD"))
SYS_STUPS77EFBJCOTDFMHM8CHP7Q1 ("VAL2","VAL3")
```

如同在接下来的查询输出中所示，隐藏列和扩展信息的名称相同。还要注意扩展信息的定义是如何添加到列上的：

```
SQL> SELECT column_name, data_type, hidden_column, data_default
  2  FROM user_tab_cols
  3  WHERE table_name = 'T'
  4  ORDER BY column_id;

COLUMN_NAME                    DATA_TYPE HIDDEN DATA_DEFAULT
```

① 无法创建超过20条扩展信息。如果你想创建超过20扩展信息，那么就会出现如下错误：ORA-20008：Number of extensions in table<talble>already reaches the upper limit (20)。

```
------------------------------ --------- ------ -----------------------------------
ID                             NUMBER    NO
VAL1                           NUMBER    NO
VAL2                           NUMBER    NO
VAL3                           NUMBER    NO
PAD                            VARCHAR2  NO
SYS_STU0KSQX64#I01CKJ5FPGFK3W9 VARCHAR2  YES    UPPER("PAD")
SYS_STUPS77EFBJCOTDFMHM8CHP7Q1 NUMBER    YES    SYS_OP_COMBINED_HASH("VAL2","VAL3")
```

警告 因为一组列的扩展统计信息来自一个散列函数（sys_op_combined_hash），所以这些统计信息只能够应用于等价谓词上。换句话说，如果使用了基于类似BETWEEN以及< 或>这样的运算符的谓词条件，则查询优化器无法利用扩展统计信息。一组列的扩展统计信息也可以用于估算GROUP BY条件的基数，并且从11.2.0.3版本开始，也可以用于DISTINCT运算符和SELECT子句。

要删除一个扩展信息，dbms_stats包提供了drop_extended_stats存储过程。在接下来的例子中，PL/SQL代码块删除了之前建立的两个扩展信息：

```
BEGIN
  dbms_stats.drop_extended_stats(ownname   => user,
                                 tabname   => 'T',
                                 extension => '(upper(pad))');
  dbms_stats.drop_extended_stats(ownname   => user,
                                 tabname   => 'T',
                                 extension => '(val2,val3)');
END;
```

完全没有必要因为一件小事就决定哪一组列适合在上面创建扩展信息。下面的方法可以用于11.2.0.2之后的版本中（在脚本seed_col_usage.sql中有完整的例子可供访问）。

(1) 调用dbms_stats包的seed_col_usage存储过程来指示查询优化器记录以下信息：WHERE子句中指定的关于谓词的信息，GROUP BY子句中引用的关于列的信息，以及从11.2.0.3版本开始起SELECT子句中关于DISTINCT运算符的信息。做该记录要么是为了sqlset_name和owner_name参数中指定的SQL调优集的所有SQL语句，要么是为了由time_limit参数指定的以秒为单位的一段时间内进行了硬解析（不需要执行，因此使用EXPLAIN PLAN语句就足够了）的所有SQL语句：

```
SQL> BEGIN
  2    dbms_stats.seed_col_usage(sqlset_name => NULL,
  3                              owner_name => NULL,
  4                              time_limit => 30);
  5  END;
  6  /
```

(2) 一旦记录过程完毕，就会调用dbms_stats包的report_col_usage函数来报告列的使用情况。每个列的使用模式都被报告出来。例如，在下面的输出中，val1和val2列都是一个基于等值条件的单表谓词的一部分：

```
SQL> SELECT dbms_stats.report_col_usage(ownname => user, tabname => 't')
  2  FROM dual;

DBMS_STATS.REPORT_COL_USAGE(OWNNAME=>USER,TABNAME=>'T')
--------------------------------------------------------------------------
```

```
LEGEND:
.......

EQ          : Used in single table EQuality predicate
RANGE       : Used in single table RANGE predicate
LIKE        : Used in single table LIKE predicate
NULL        : Used in single table is (not) NULL predicate
EQ_JOIN     : Used in EQuality JOIN predicate
NONEQ_JOIN  : Used in NON EQuality JOIN predicate
FILTER      : Used in single table FILTER predicate
JOIN        : Used in JOIN predicate
GROUP_BY    : Used in GROUP BY expression
...............................................................................

###############################################################################

COLUMN USAGE REPORT FOR CHRIS.T
...............................

1. VAL1                                : EQ
2. VAL2                                : EQ
3. VAL3                                : EQ
4. (VAL1, VAL2)                        : FILTER
5. (VAL1, VAL3)                        : FILTER
6. (VAL2, VAL3)                        : GROUP_BY
###############################################################################
```

(3) 使用dbms_stats包的create_extended_stats存储过程来创建扩展信息。注意扩展信息自身的定义如果不是作为参数来传递，那么定义就从记录过程中存储的信息中获得。因此，只需要模式和表名两个参数。请注意，在下面的例子中如何调用一次create_extended_stats函数就创建三个扩展：

```
SQL> SELECT dbms_stats.create_extended_stats(ownname => user, tabname => 't')
  2  FROM dual;

DBMS_STATS.CREATE_EXTENDED_STATS(OWNNAME=>USER,TABNAME=>'T')
--------------------------------------------------------------------------------
################################################################################

EXTENSIONS FOR CHRIS.T
......................

1. (VAL1, VAL2)                        : SYS_STU4K1K3JNH1Z9#_L_V93K3DT4 created
2. (VAL1, VAL3)                        : SYS_STUS574STTDWYBF6PGQN#XHGGJ created
3. (VAL2, VAL3)                        : SYS_STUPS77EFBJCOTDFMHM8CHP7Q1 created
################################################################################
```

(4) 在创建完扩展信息后，重新收集修正过的表的对象统计信息。

在12.1版本中，扩展信息也可以由数据库引擎自动创建。实际上，对于利用统计信息反馈的SQL语句，查询优化器可以创建一个用于通知数据库引擎创建扩展信息的SQL计划指令（SQL plan directive）。这样，就可以避免将来在统计信息反馈过程中的重新优化。完整的例子可以在脚本seed_col_usage.sql中找到。这里有两个关键点需要了解。第一，扩展信息可以创建并且会自动创建。第二，扩展信息只有在对象统计信息已经收集的情况下才可创建。换句话说，创建SQL计划指令和创

建扩展信息之间的时间间隔依赖于对象统计信息收集的频率。

有意思的是，要注意扩展统计信息以另一个特性为基础，这个特性是在11.1版本中引入的，被称作虚拟列（virtual column）。虚拟列是不存储数据而只简单地通过基于其他列的表达式来生成其内容的列。这在应用程序频繁使用某个给定的表达式时非常有用。典型的例子是，在一个VARCHAR2列上应用upper函数，或者在一个DATE列上应用trunc函数。如果这些表达式的使用非常频繁，那么像下面这样直接在表上定义这些表达式就非常合理了：

```
SQL> CREATE TABLE persons (
  2    name VARCHAR2(100),
  3    name_upper AS (upper(name))
  4  );

SQL> INSERT INTO persons (name) VALUES ('Michelle');

SQL> SELECT name
  2  FROM persons
  3  WHERE name_upper = 'MICHELLE';

NAME
----------
Michelle
```

在第13章中会看到虚拟列上同样可以建立索引。

虚拟列的主要问题是，与扩展统计信息相比，它们会改变某些SQL语句的行为（例如，SELECT *语句和没有列清单的INSERT语句），除非它们被定义成不可见的（虚拟列的可见性自12.1版本起可设置）。换句话说，因为扩展统计信息是基于隐藏列的，它们对于应用程序来说是完全透明的。

无论虚拟列是如何定义的（不管是通过用户显式定义还是通过扩展统计信息隐式定义），关于它们的对象统计信息和直方图都会正常收集，认识到这一点非常重要。这样一来，查询优化器就获得了关于数据的额外统计信息。

SQL计划指令

SQL计划指令是在12.1版本中引入的新概念。它们的用途是帮助查询优化器应对错误的估算。要达到这个目的，SQL计划指令将引起错误估算的表达式信息存储在数据字典中。因为它们并不与具体的SQL语句相关联，所以不仅多个SQL计划指令可以同时应用于一个单独的SQL语句，而且一个单独的SQL计划指令也可以应用于多个SQL语句。

在某些情况下，SQL计划指令通知数据库引擎自动创建扩展统计信息（明确地说，列组）。如果无法创建扩展统计信息，则会通知查询优化器使用动态采样。

当初始化参数optimizer_adaptive_features的值为TRUE（即默认值）时会启用SQL计划指令。激活SQL计划指令时，数据库引擎会自动维护（例如，创建和清除）SQL计划指令。一些管理操作也可以通过dbms_spd包手动执行。

可用的SQL计划指令信息可以通过dba_sql_plan_directives和dba_sql_plan_dir_objects视图来查询（这些视图的cdb版本也可以使用）。

8.2.5 索引统计信息

在介绍索引统计信息之前，我们来根据图8-5简要回顾一下索引的结构。处于顶端的数据块称为根块（root block）。这个块就是每次查询的起始块。根块又引用分支块（branch block）。注意，也可以将根块看作一个分支块。每个分支块又相应地引用另一级别的分支块，或者如图8-5所示，引用叶子块（leaf block）。叶子块存储键值（在本例中，键值是在6到89之间的一些数字），并存储引用数据的rowid。对于任何一个给定的索引，根块和每个叶子块之间的分支块的数量永远是相同的。换句话说，索引永远是平衡的。注意，为支持高效率的范围值查找（例如，在25和45之间的所有值），叶子块都互相链接起来。

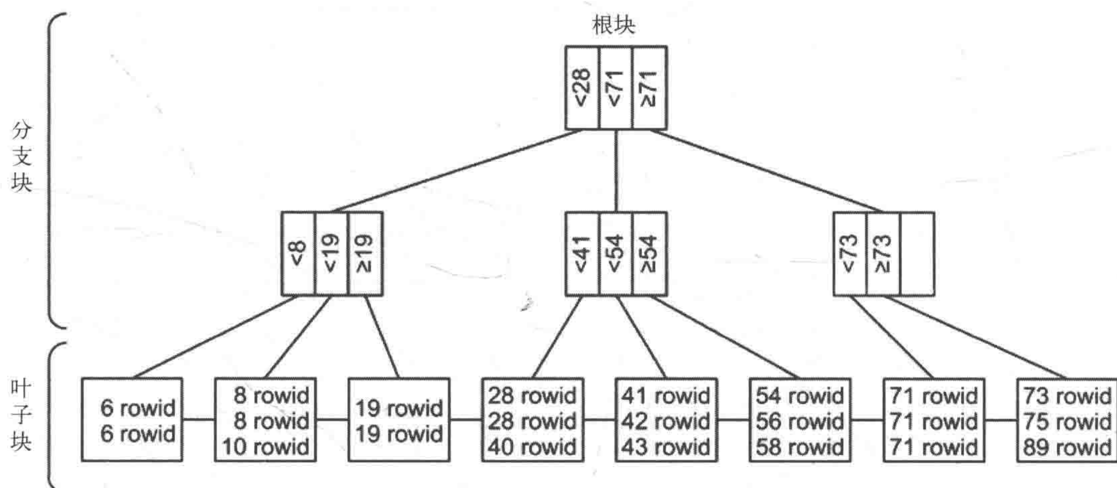


图8-5 基于B+-tree的索引结构

并非所有索引都具有这三种类型的块。实际上，分支块只有在根块无法存储引用的所有叶子块时才会出现。此外，如果索引非常小，那么它会由一个单独的块组成，并包含通常由根块和叶子块存储的所有数据。

下面的查询展示了如何获取一张表最重要的索引统计信息：

```
SQL> SELECT index_name AS name,
  2         blevel,
  3         leaf_blocks AS leaf_blks,
  4         distinct_keys AS dst_keys,
  5         num_rows,
  6         clustering_factor AS clust_fact,
  7         avg_leaf_blocks_per_key AS leaf_per_key,
  8         avg_data_blocks_per_key AS data_per_key
  9  FROM user_ind_statistics
 10  WHERE table_name = 'T';

NAME     BLEVEL LEAF_BLKS DST_KEYS NUM_ROWS CLUST_FACT LEAF_PER_KEY DATA_PER_KEY
-------- ------ --------- -------- -------- ---------- ------------ ------------
T_PK          1         2     1000     1000        979            1            1
T_VAL1_I      1         2      431      497        478            1            1
T_VAL2_I      1         3        6     1000        175            1           29
```

这个查询返回的索引统计信息如下所示。

- blevel是为了访问叶子块而需要读取的分支块的数量，包含根块在内。
- leaf_blocks是索引的叶子块数量。
- distinct_keys是索引中不重复键值的数量。
- num_rows是索引中键值的数量。对于主键，这个值与distinct_keys相等。
- clustering_factor表明有多少相邻的索引条目没有指向表中相同的数据块。如果表和索引存储数据的顺序相类似，则群集因子（clustering factor）较低。其最小值是表中非空数据块的数量。如果表和索引存储数据的顺序不同，则群集因子较高。其最大值是索引中键值的数量。我会在第13章中详细讨论这个值的计算方式以及它对性能的影响。有必要提一下，对于bitmap索引，不会计算实际意义上的群集因子。实际上，会将其值设置为索引的键值数量。
- avg_leaf_blocks_per_key是存储一个单独的键值所需的平均叶子块数量。这个值是使用公式8-2通过其他的统计信息计算得来的。

  公式8-2　计算存储一个单独的键值所需叶子块的平均数量

$$avg_leaf_blocks_per_key \approx \frac{leag_blocks}{distinct_keys}$$

- avg_data_blocks_per_key是在表中某个单独的键值所引用的数据块平均数量。这个值是使用公式8-3通过其他的统计信息计算得来的。

  公式8-3　计算某个单独的键值所引用的数据块平均数量

$$avg_data_blocks_per_key \approx \frac{clustering_factor}{distinct_keys}$$

8.2.6　分区对象统计信息

分区对象是由段的集合组成的逻辑概念。举例来说，下面的SQL语句创建一张拥有16个段的分区表，如图8-6所示。这16个段是在表空间中实际存储数据的对象，四个分区和表仅是元数据对象。它们只存在于数据字典中：

```
CREATE TABLE t (id NUMBER, tstamp DATE, pad VARCHAR2(1000))
PARTITION BY RANGE (tstamp)
SUBPARTITION BY HASH (id)
SUBPARTITION TEMPLATE
(
  SUBPARTITION sp1,
  SUBPARTITION sp2,
  SUBPARTITION sp3,
  SUBPARTITION sp4
)
(
  PARTITION q1 VALUES LESS THAN (to_date('2014-04-01','YYYY-MM-DD')),
  PARTITION q2 VALUES LESS THAN (to_date('2014-07-01','YYYY-MM-DD')),
  PARTITION q3 VALUES LESS THAN (to_date('2014-10-01','YYYY-MM-DD')),
  PARTITION q4 VALUES LESS THAN (to_date('2015-01-01','YYYY-MM-DD'))
)
```

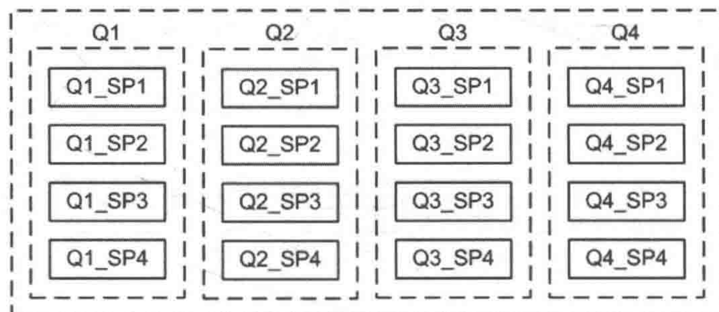


图8-6　拥有16个段的范围-散列分区表

对于分区的对象，数据库引擎分别能够在表/索引级别上及分区和子分区级别上处理前面章节讨论的所有对象统计信息（换句话说，表统计信息、列统计信息、直方图和索引统计信息）。在所有级别上拥有统计信息是有必要的，因为根据所处理的SQL语句，查询优化器着重访问最能够描述段的对象统计信息。简言之，仅在解析阶段查询优化器可以确定是否访问某个特定的分区或者子分区时，查询优化器才使用分区和子分区统计信息。否则，查询优化器通常会使用表/索引级别统计信息。（但在某些情况下，查询优化器在同一时刻既使用表/索引级别统计信息，也使用分区和子分区级别统计信息。）

8.3　收集对象统计信息

为收集对象统计信息，dbms_stats包含有多个存储过程。使用多个存储过程是因为，根据不同的情形，收集对象统计信息的处理过程应该发生在整个数据库、数据字典、模式或者单独的表级别上。

- gather_database_stats为整个数据库收集对象统计信息。
- gather_dictionary_stats为数据字典收集对象统计信息。注意，数据字典不仅是由存储在sys模式下的对象组成，同时也包括由Oracle为可选组件安装的其他模式下的对象。
- gather_fixed_objects_stats为称作固定表（又称为x$表）和固定索引的特殊对象收集对象统计信息，它们是数据字典的组成部分。固定表，通常用于动态性能视图中，是仅存在于内存中的结构。基于这个原因，需要对它们进行特殊处理。要想知道这个过程与哪些表有关系，可以使用下面的查询。注意，并没有为所有的固定表收集对象统计信息：

```
SELECT name
FROM v$fixed_table
WHERE type = 'TABLE'
```

- gather_schema_stats为整个模式收集对象统计信息。
- gather_table_stats为表收集包括列在内的对象统计信息，还可以为其索引收集统计信息。
- gather_index_stats为索引收集对象统计信息。

注意　dbms_stats包并不是收集对象统计信息的唯一特性。实际上，CREATE INDEX和ALTER INDEX语句在创建索引时会自动收集对象统计信息。此外，从12.1版本开始，CTAS语句和将数据插入到空表中的直接路径插入也会自动收集对象统计信息。要知道由dbms_stats包计算的对象统计信息要优先于自动收集的统计信息。因此，不能在任何情况下都总是依赖自动收集统计信息。

---

dbms_stats包提供的存储过程接受的不同参数可以分为三种主要的类型。通过第一组参数可以指定目标对象，通过第二组可以指定收集的选项，而通过第三组可以指定是否在覆盖当前统计信息之前备份它们。表8-3总结了在各个存储过程中可用的不同参数。接下来的三个小节将详细描述每个参数的使用范围和用法。

表8-3　用于收集对象统计信息的存储过程的参数

参　数	数据库	数据字典	固定对象	模　式	表	索　引
目标对象						
ownname				✓	✓	✓
indname						✓
tabname					✓	
partname					✓	✓
comp_id		✓				
granularity	✓	✓		✓	✓	✓
cascade	✓	✓		✓	✓	
gather_fixed	✓			✓		
gather_sys	✓					
gather_temp	✓			✓		
options	✓	✓		✓	✓*	
objlist	✓	✓		✓		
force				✓	✓	✓
obj_filter_list	✓	✓		✓		
收集选项						
estimate_percent	✓	✓		✓	✓	✓
block_sample	✓	✓		✓	✓	
method_opt	✓	✓		✓	✓	
degree	✓	✓		✓	✓	✓
no_invalidate	✓	✓	✓	✓	✓	✓
备份表						
stattab	✓	✓	✓	✓	✓	✓
statid	✓	✓	✓	✓	✓	✓
statown	✓	✓	✓	✓	✓	✓

* 表示从12.1版本起开始可用。

8.3.1　目标对象

目标对象参数指定要为哪些对象收集对象统计信息。

- ownname指定要处理的模式的名称。这个参数是强制参数。
- indname指定要处理的索引的名称。这个参数是强制参数。
- tabname指定要处理的表的名称。这个参数是强制参数。

- partname指定要处理的分区或者子分区的名称。如果没有指定任何值，则可能会收集所有分区和子分区的对象统计信息，具体取决于granularity参数（见下面）的取值。默认值为NULL。
- comp_id指定要处理的组件的ID。因为组件的ID无法用于收集统计信息，所以会在内部将它转换成一组模式的列表。要想知道对于一个给定的组件都处理了哪些模式，可以使用下面的查询[①]。注意这个查询的输出受多个因素的影响，比如版本和实际安装的组件等。sys和system模式独立于此参数，总是会被处理。如果指定了非法值，则不会返回错误信息，并且sys和system模式会正常进行处理。通过使用默认值NULL，所有的组件都会被处理：

```
SQL> SELECT u.username AS schema_name, r.cid AS comp_id, r.cname AS comp_name
  2  FROM dba_users u,
  3       (SELECT schema#, cid, cname
  4        FROM sys.registry$
  5        WHERE status IN (1, 3, 5)
  6        AND namespace = 'SERVER'
  7        UNION ALL
  8        SELECT s.schema#, s.cid, cname
  9        FROM sys.registry$ r, sys.registry$schemas s
 10        WHERE r.status IN (1,3,5)
 11        AND r.namespace = 'SERVER'
 12        AND r.cid = s.cid) r
 13  WHERE u.user_id = r.schema#
 14  ORDER BY r.cid, u.username;

SCHEMA_NAME        COMP_ID COMP_NAME
------------------ ------- -----------------------------------
SYS                APS     OLAP Analytic Workspace
SYS                CATALOG Oracle Database Catalog Views
SYS                CATJAVA Oracle Database Java Packages
APPQOSSYS          CATPROC Oracle Database Packages and Types
DBSNMP             CATPROC Oracle Database Packages and Types
DIP                CATPROC Oracle Database Packages and Types
GSMADMIN_INTERNAL  CATPROC Oracle Database Packages and Types
ORACLE_OCM         CATPROC Oracle Database Packages and Types
OUTLN              CATPROC Oracle Database Packages and Types
SYS                CATPROC Oracle Database Packages and Types
SYSTEM             CATPROC Oracle Database Packages and Types
CTXSYS             CONTEXT Oracle Text
SYS                JAVAVM  JServer JAVA Virtual Machine
LBACSYS            OLS     Oracle Label Security
MDSYS              ORDIM   Oracle Multimedia
ORDDATA            ORDIM   Oracle Multimedia
ORDPLUGINS         ORDIM   Oracle Multimedia
ORDSYS             ORDIM   Oracle Multimedia
SI_INFORMTN_SCHEMA ORDIM   Oracle Multimedia
WMSYS              OWM     Oracle Workspace Manager
MDSYS              SDO     Spatial
ANONYMOUS          XDB     Oracle XML Database
XDB                XDB     Oracle XML Database
```

① 遗憾的是，Oracle并不会使所有必需的信息都在数据字典中可见。所以，这个查询是基于内部表的。系统权限select any dictionary能够提供对必要的表的访问权限。

```
XS$NULL              XDB     Oracle XML Database
SYS                  XML     Oracle XDK
SYS                  XOQ     Oracle OLAP API
```

- granularity指定会在哪个级别处理已分区对象的统计信息。这个参数接受表8-4中的值。默认值是auto（默认值可以修改，参见8.4节）。关于管理已分区对象的对象统计信息的详细信息，请参考8.7节。

表8-4 granularity参数接受的参数

值	含　义
all	收集表/索引、分区以及子分区的统计信息
auto	收集表/索引和分区的统计信息。只有当表使用列表或范围分区时才会收集子分区的统计信息
global	只收集表/索引的统计信息
global and partition	收集表/索引和分区的统计信息
approx_global and partition	与global and partition类似，但是在表/索引级别使用提取的统计信息。在10.2.0.5版本以及11.1.0.7之后的版本中可用
partition	只收集分区的统计信息
subpartition	只收集子分区的统计信息

- cascade指定是否处理索引的数据。这个参数接受的值为TRUE、FLASE以及dbms_stats.auto_cascade。后者是一个设置为NULL的常量值，让数据库引擎来决定是否收集索引统计信息。默认值是dbms_stats.auto_cascade（默认值可以修改，参见8.4节）。
- gather_fixed指定是否为固定表收集对象统计信息。这个参数接受的值为TRUE和FALSE。默认值是FALSE。
- gather_sys指定是否收集sys模式下的数据。这个参数接受的值为TRUE和FALSE。默认值是FALSE。
- gather_temp指定是否收集临时表的数据。这个参数接受的值为TRUE和FALSE。默认值是FALSE。参见8.5节以获取更多详细信息。
- options指定有哪些对象以及是否收集它们。这个参数接受的值在表8-5中列出。但是，当这个参数与gather_table_stats存储过程一起使用时，只有gather和gather auto受支持。默认值是gather。

表8-5 options参数接受的值

值	含　义
gather	处理所有对象
gather auto	让存储过程不仅决定要处理哪些对象而且还要决定如何去处理这些对象。当在gather_table_stats存储过程中使用参数的值为not时，除了ownname、objlist、stattab、statid以及statown以外的所有参数都会被忽略
gather stale	只有包含过期对象统计信息的对象会被处理。注意，不会将没有对象统计信息的对象视为过期的
gather empty	只有没有对象统计信息的对象才会被处理
list auto	对于所列举的对象会按照gather auto选项进行处理
list stale	对于所列举的对象会按照gather stale选项进行处理
list empty	对于所列举的对象会按照gather empty选项进行处理

对象统计信息的过期

为识别出对象统计信息是否过期，数据库引擎对每张表上通过SQL语句修改的数据行的数量进行计数（约计）。计数的结果可以通过数据字典视图all_tab_modifications、dba_tab_modifications（这个视图仅从11.2版本开始才可用）、user_tab_modifications来查看，还可以通过12.1的多租户环境下的cdb_tab_modifications视图来查看。下面的查询是一个样例：

```
SQL> SELECT inserts, updates, deletes, truncated
  2  FROM user_tab_modifications
  3  WHERE table_name = 'T';

INSERTS UPDATES DELETES TRUNCATED
------- ------- ------- ---------
    775   14200      66 NO
```

根据这个信息，dbms_stats包能够确定与某个对象关联的对象统计信息是否过期。在10.2版本中，必须有至少10%的数据行被修改了才会认为对象统计信息过期。从11.1版本开始，可以通过stale_percent首选项来配置这个阈值。默认值是10%。8.5节中会介绍如何修改这个值。

要小心，因为在10.2.0.5、11.2.0.1以及11.2.0.2版本中，通过Data Pump导入到一张空表中的数据的计数和正常插入的数据相比是不正确的。因此，在导入之后，会认为对象统计信息过期。

计数是由数据库端的初始化参数statistics_level控制的。如果这个参数设置为typical（也就是默认值）或者all，那么计数为启用状态。

- objlist根据options参数取值的不同，返回已经处理过的或者即将要处理的对象的列表。这是一个基于dbms_stats包中定义的类型的输出参数。举个例子，下面的PL/SQL代码块展示了如何显示处理过的对象列表：

```
SQL> DECLARE
  2    l_objlist dbms_stats.objecttab;
  3    l_index PLS_INTEGER;
  4  BEGIN
  5    dbms_stats.gather_schema_stats(ownname => 'HR',
  6                                   objlist => l_objlist);
  7    l_index := l_objlist.FIRST;
  8    WHILE l_index IS NOT NULL
  9    LOOP
 10      dbms_output.put(l_objlist(l_index).ownname || '.');
 11      dbms_output.put_line(l_objlist(l_index).objname);
 12      l_index := l_objlist.next(l_index);
 13    END LOOP;
 14  END;
 15  /
HR.COUNTRIES
HR.DEPARTMENTS
HR.EMPLOYEES
HR.JOBS
HR.JOB_HISTORY
HR.LOCATIONS
HR.REGIONS
```

- force指定是否覆盖已锁定的统计信息。如果将这个参数设置为FALSE，而一个用来处理一张单独的表或者索引的存储过程正在处理锁定的统计信息，那么就会引发一个错误（ORA-20005）。这个参数接受的值为TRUE和FALSE。可以在8.10节中找到更多关于锁定的统计信息的内容。
- obj_filter_list用于指定只为那些至少满足作为参数传递的其中一个过滤条件的对象收集统计信息。它基于dbms_stats包自身中定义的objecttab类型，并且只在11.1及以后的版本中才可用。下面的PL/SQL代码块展示了如何为HR模式下的所有表和SH模式下的所有表以及使用字母C开头的对象收集统计信息：

```
DECLARE
  l_filter dbms_stats.objecttab := dbms_stats.objecttab();
BEGIN
  l_filter.extend(2);
  l_filter(1).ownname := 'HR';
  l_filter(2).ownname := 'SH';
  l_filter(2).objname := 'C%';
  dbms_stats.gather_database_stats(obj_filter_list => l_filter,
                                   options         => 'gather');
END;
```

8.3.2　收集选项

在表8-2中列出的收集选项参数指定了收集统计信息的过程如何进行，收集哪些类型的列统计信息，以及是否使从属SQL游标失效。各个选项如下所示。

- estimate_percent指定收集统计信息时是否使用采样。有效值为0.000001到100之间的十进制数字。当值为100时，其含义与NULL一样，意味着不进行采样。常量dbms_stats.auto_sample_size，也就是默认值（这个默认值可以修改，参见8.4节），会让存储过程来决定采样大小。从11.1版本开始，推荐使用这个值。实际上，在大多数情况下，使用这个默认值不仅使收集的统计信息比使用类似10%的采样率进行收集要更加精确，而且也更加快速。这是因为在11.1版本中引入了一种全新的算法，而且这种算法只能在指定参数值为dbms_stats.auto_sample_size时才可以使用。同时也要指出，因为这个新的算法需要对收集统计信息的表执行全表扫描，这在磁盘I/O子系统相对缓慢的系统上可能会花费很长时间。还要注意，某些特性（高频率直方图、混合直方图，还有增量统计信息）要求指定dbms_stats.auto_sample_size。有一点很重要，将一个十进制数字作为参数传递进来时，由参数estimate_percent指定的值只不过是用于收集统计信息的最小百分比。事实上，正如下例所示，如果dbms_stats包认为由estimate_percent指定的值过小，那么程序包可能会自动增大这个值。假如不使用dbms_stats.auto_sample_size进行收集，可以使用较小的百分比来加速对象统计信息的收集；一般来说小于10个百分点就比较合适。对于特大的表，0.5个百分点、0.1个百分点，或者更小的值也不会有问题。实际上的最佳值取决于数据分布情况。如果不确定该选什么，干脆就尝试不同的估算百分比然后比较收集的统计信息。这样，你可能会在性能和精确度之间找到一个最佳折衷点。注意，使用小的估算百分比可能不会产生稳定的统计信息。因为如果收集统计信息是在数据库或模式级别上执行，则过小的值会被自动增大，估算的百分比应该按最大的那张表来选择。顺便提一下，在外部表上采样是不受支持的：

```
SQL> BEGIN
  2    dbms_stats.gather_schema_stats(ownname          => user,
  3                                   estimate_percent => 0.5);
  4  END;
  5  /

SQL> SELECT table_name, sample_size, num_rows,
  2         round(sample_size/num_rows*100,1) AS "%"
  3  FROM user_tables
  4  WHERE num_rows > 0
  5  ORDER BY table_name;

TABLE_NAME             SAMPLE_SIZE NUM_ROWS      %
---------------------- ----------- -------- ------
CAL_MONTH_SALES_MV              48       48  100.0
CHANNELS                         5        5  100.0
COSTS                         4975    81391    6.1
COUNTRIES                       23       23  100.0
CUSTOMERS                     5435    55002    9.9
FWEEK_PSCAT_SALES_MV          4742    11001   43.1
PRODUCTS                        72       72  100.0
PROMOTIONS                     503      503  100.0
SALES                         4639   927800    0.5
SALES_TRANSACTIONS_EXT      916039   916039  100.0
TIMES                         1826     1826  100.0
```

- block_sample指定是将行级采样还是块级采样用于统计信息的收集过程。尽管行级采样更加精确，但是块级采样更加迅速。因此，只有确定数据是随机分布时才应该使用块级采样。这个参数接受的值为TRUE和FALSE。默认值是FALSE。自11.1版本开始，这个参数只有在estimate_percent参数的值没有设置为dbms_stats.auto_sample_size时才会出现。
- method_opt指定是否收集列统计信息和直方图以及如何收集它们。下面是三种典型的用例[1]。
  - 为所有列[2]收集列统计信息和直方图。所有直方图都按照相同的size_clause参数值创建。如果指定的值为1，则不会创建任何直方图。语法如图8-7所示。举个例子，通过使用值for all columns size 254，会为每个列创建一个拥有最高254个桶的直方图。

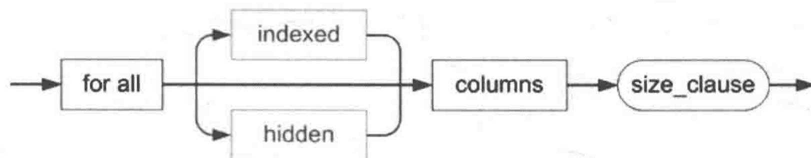


图8-7　使用具有单一取值的size_clause参数为所有列收集列统计信息和直方图（见表8-5）

① 为简单起见，我不会描述所有的可能性，因为其中许多要么是冗余的，要么是在实践中应用受限的。

② 实际上，通过选项indexed和hidden，可以限制仅为索引列和隐藏列收集统计信息。一般来说，对象统计信息应该对所有列都可用。基于这个原因，应该避免使用这两个选项（因此在图8-7中它们被标记为灰色）。如果对于某些列来说不需要对象统计信息，则应该转而使用图8-8中描述的语法。使用hidden选项确实有一个合理的理由，那就是为某个作为扩展而刚刚加入到表中的虚拟列收集统计信息。

- 在所有列上收集列统计信息并在一部分列上收集直方图。所有直方图都按照相同的size_clause参数值创建。语法为图8-7和图8-8的一个组合：前者指定列统计信息的收集，后者指定直方图的收集。举个例子，指定"for all columns size 1 for columns size 254 col1"会使得系统在每个列上收集列统计信息并只在col1列上收集一个最多具有254个桶的直方图。

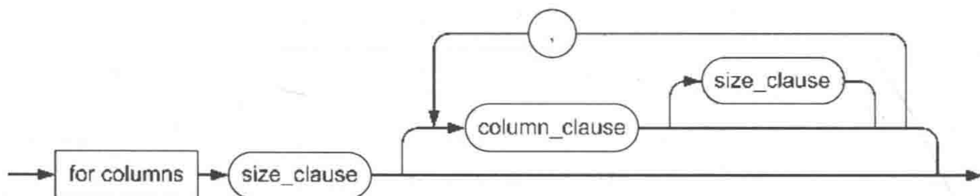


图8-8　只为一部分列收集列统计信息和直方图，但为size_clause参数使用不同的值（见表8-6）。对于没有明确指明size_clause参数的列，使用默认的size_clause参数（即本图中左边的那个）。如果没有指定列，则不会收集任何列统计信息。column_clause参数可以是一个列名、一个扩展名或者一个扩展信息。如果指定了一个不存在的扩展信息，就会自动创建新的扩展信息。这个语法只能在调用gather_table_stats存储过程时才有效

表8-6　size_clause参数接受的值

值	含　义
size n	指明最大的桶数量。如果指定了size 1，则不会创建直方图。但无论怎样，列统计信息都是正常收集的
size skewonly	只为含有歪斜数据的列收集直方图。桶的数量由系统自动确定
size auto	只为含有歪斜数据的列收集直方图，就像skewonly一样，此外，还为那些在WHERE条件中被引用的列收集直方图。第二个条件基于列使用的历史信息。桶的数量由系统自动确定
size repeat	刷新可用的直方图

- 只为一部分列收集列统计信息和直方图，并且为size_clause参数使用不同的值。语法如图8-8所示。举例来说，指定for columns size 1 id, col1 size 100, col2 size 5, col3会使得系统在四个列上收集列统计信息，但是只会在列col1和col2上分别收集最多有100个桶和5个桶的直方图。

这个参数的默认值是for all columns size auto（默认值可以修改，参见8.4节）。为简单起见，使用size skewonly或者size auto。如果执行得太慢或者选择的桶数不合理（或者所需要的直方图根本没有创建），那么就手工指定列的列表。如果指定了NULL，则会使用for all columns size 1。

列使用历史

dbms_stats包依赖于列使用历史来决定哪些列的直方图是有帮助的。为收集历史，当产生一个新的执行计划时，查询优化器会跟踪哪些列被WHERE子句引用了，并会存储它在SGA中找到的信息。然后，每隔一定时间，数据库引擎会将这些信息存储在数据字典表col_usage$中。通过执行类似下面这样的基于内部数据字典表的查询（该查询可以在col_usage.sql脚本中找到），就可

以知道哪些列被WHERE子句引用了以及使用的是哪种类型的谓词。timestamp列表明了最近使用的时间。其他列为硬解析次数（实际上，提供相同信息并接连不断执行的硬解析不包括在内）的计数。从未被WHERE子句引用过的列不会出现在col_usage$表中，所以，在输出中除了name之外，其他列值都为空。

```
SQL> SELECT c.name, cu.timestamp,
  2         cu.equality_preds AS equality, cu.equijoin_preds AS equijoin,
  3         cu.nonequijoin_preds AS noneequijoin, cu.range_preds AS range,
  4         cu.like_preds AS "LIKE", cu.null_preds AS "NULL"
  5  FROM sys.col$ c, sys.col_usage$ cu, sys.obj$ o, dba_users u
  6  WHERE c.obj# = cu.obj# (+)
  7  AND c.intcol# = cu.intcol# (+)
  8  AND c.obj# = o.obj#
  9  AND o.owner# = u.user_id
 10  AND o.name = 'T'
 11  AND u.username = user
 12  ORDER BY c.col#;

NAME TIMESTAMP EQUALITY EQUIJOIN NONEEQUIJOIN RANGE  LIKE  NULL
---- --------- -------- -------- ------------ ----- ----- -----
ID   27-MAY-14        1        1            0     0     0     0
VAL1 27-MAY-14        1        0            0     0     0     0
VAL2
VAL3 27-MAY-14        1        1            0     0     0     0
PAD  27-MAY-14        0        0            0     1     0     0
```

自11.2.0.2版本开始,dbms_stats包的report_col_usage函数使得对col_usage$信息的选择变得更加容易了。注意，这是在8.2.4节中讨论过的相同函数。但是要知道，如果没有使用seed_col_usage函数，report_col_usage函数返回的报告不会包含关于潜在列组的信息。下面的查询展示了一个例子，并截取了一段输出：

```
SQL> SELECT dbms_stats.report_col_usage(ownname => user, tabname => 't')
  2  FROM dual;

COLUMN USAGE REPORT FOR CHRIS.T
...............................

1. ID                                  : EQ EQ_JOIN
2. PAD                                 : RANGE
3. VAL1                                : EQ
4. VAL3                                : EQ EQ_JOIN
```

此外，从11.2.0.2版本开始，dbms_stats包还提供了一种重置col_usage$表内容的方法。可以通过使用reset_col_usage存储过程来达到此目的。

- degree指定为一个单独对象收集统计信息所使用的并行度。要使用在表/索引级别定义的并行度，请将这个值指定为NULL。要让存储过程自行决定并行度，指定这个值为常量dbms_stats.default_degree。其默认值为NULL（这个默认值可以修改，参见8.4节）。注意处理

多个对象时是串行执行的，除非使用了并行统计信息收集。这意味着并行化只对加速大型对象统计信息的收集起作用。要在同时处理多个对象时使用并行化，则有必要进行手工并行化（也就是说，同时开启多个任务）。参考第15章获取更多关于并行处理的详细信息。并行收集对象统计信息只在企业版中可用。

并行统计信息收集

默认情况下，dbms_stats包只会并行化在表或者分区级别（根据degree参数）的收集过程。也就是说，在任意给定的时间点，只会处理一个单独的表或分区。如果数据库服务器拥有大量空闲资源，而且需要处理的表或分区很多，这种情况下同时处理它们或许比较合理。基于这个目的，自11.2.0.2版本开始，Oracle Database提供了一种称为**并行统计信息收集**（concurrent statistics gathering）的新的收集模式。

并行统计信息收集是在gather_*_stats存储过程中实现的。要控制它，可以使用concurrent首选项。根据你所运行的版本，可以将它设置为下列值。

- 11.2：设置为FALSE会禁用这个特性（这是默认值），反之，设置为TRUE会启用这个特性。
- 12.1：设置为OFF会禁用这个特性（这是默认值），设置为MANUAL则只为手动统计信息收集启用这个特性，设置为AUTOMATIC只为自动统计信息收集启用这个特性，而设置为ALL会为所有类型的统计信息收集启用这个特性。

要想利用并行统计信息收集，必须满足以下要求。

- 初始化参数job_queue_processes的值至少应设置为4。这是因为同时处理多个表或者分区时，dbms_stats包会向Scheduler提交一定数量的任务。
- Resource Manager应该是启用状态。因为dbms_stats包并不会控制同时运行多少个并发任务，所以，如果没有Resource Manager，系统的负载可能会超出控制范围。实际上，并行统计信息收集依赖于Scheduler和Resource Manager来生成最佳负载。
- 提交收集任务的用户必须拥有dba角色或者拥有以下权限：CREATE JOB、MANAGE SCHEDULER以及MANAGE ANY QUEUE。

- no_invalidate指定是否使依赖于所处理对象的游标失效，并进而指定是否禁止这些游标在未来继续使用。这个参数接受的值为TRUE、FALSE以及dbms_stats.auto_invalidate。将这个参数设置为TRUE时，依赖于更改的对象统计信息的游标不会失效，因此在未来的执行中仍然可以继续使用这些游标。而另一方面，如果将它设置为FALSE，所有相关的游标会立即失效。如果使用值dbms_stats.auto_invalidate（也就是一个等于NULL的常量），那么相关的游标会在一段时间后失效。最后一种选项有利于避免重新解析的集中出现。默认值是dbms_stats.auto_invalidate（这个默认值可以修改，参见8.4节）。

  使用了dbms_stats.auto_invalidate时，dbms_stats包会将所有依赖于变更的统计信息的游标标记为延迟失效状态。程序包会在游标级别设置一个时间戳，指明这个游标何时不应该再使用了。注意这个时间戳对于每个游标都是不同的，它基于一个从游标被标记的那一刻开始最长五个小

时的随机值来设置。真实的失效动作是由服务器进程来执行的，该进程尝试重用标记为延迟失效的游标。因此，如果一个标记为延迟失效的游标从来没有被重新解析过，那么它永远不会失效。唯一的例外是关联到并行SQL语句的游标。这样的游标会通过dbms_stats包立即失效。

8.3.3　备份表

用于收集对象统计信息的所有存储过程都支持表8-2中列出的备份表参数。这些参数指示dbms_stats包在使用数据字典中新的统计信息覆盖旧的统计信息之前，将现有的统计信息备份到一张备份表中。这些参数如下所示。

- stattab指定在数据字典之外的一张备份表用于存储统计信息。如果指定了NULL（默认值），则不会使用备份表。
- statid是一个可选ID，用于识别存储在stattab参数所指定的备份表中的多组不同对象统计信息。只有合法的Oracle标识符[1]才受支持。如果指定了NULL（默认值），则不会有ID和对象统计信息关联。
- statown指定由stattab参数指定的表的所有者。默认值是NULL，此时会使用当前用户作为所有者。

要创建备份表，可以使用dbms_stats包提供的create_stat_table存储过程。如下例所示，创建备份表就是指定所有者（使用ownname参数）和备份表名称（使用stattab参数）的问题。此外，可选的tblspace参数指定将表创建在哪个表空间上。如果tblspace参数没有指定，默认情况下，备份表最终会创建在用户的默认表空间中：

```
dbms_stats.create_stat_table(ownname  => user,
                             stattab  => 'MYSTATS',
                             tblspace => 'USERS')
```

因为备份表用来存储各种不同的信息，它的大部分列是通用的。举例来说，在11.2版本中有12个列用于存储数字值（命名为n1...n12），5个列用于存储字符串值（命名为c1...c5），2个列用于存储位串值（命名为r1和r2），还有1个列存储日期时间值（命名为d1）。

请注意，这么多年以来，备份表的结构已经发生了改变。因此，你需要在升级到一个新版本的数据库后或在不同的数据库版本之间移动备份表时也升级你的备份表。否则，可能无法使用备份表。基于这个目的， dbms_stats包提供了upgrade_stat_table存储过程。要使用它你需要指定所有者（通过ownname参数）和备份表的名称（通过stattab参数）。例如：

```
dbms_stats.upgrade_stat_table(ownname => user,
                              stattab => 'MYSTATS')
```

要删除备份表，可使用dbms_stats包提供的drop_stat_table存储过程：

```
dbms_stats.drop_stat_table(ownname => user,
                           stattab => 'MYSTATS')
```

也可以通过常规DROP TABLE语句来删除备份表。

---

① 参考Oracle官方文档的*SQL Language Reference*手册中关于标识符的定义。

8.4 配置 dbms_stats 包

dbms_stats包提供了两组子程序，用来配置在之前章节中描述的某些参数的默认值。第一组仅应在10.2版本中使用。实际上，第一组子程序在11.1版本中已废弃。因此，从11.1版本开始，应该使用由第二组子程序提供的子程序。

8.4.1 传统方式

在10.2版本中，你可以更改cascade、estimate_percent、degree、method_opt、no_invalidate和granularity参数的全局默认值。这些默认值能进行修改是因为它们不是硬编码写在存储过程中的，而是在运行时从数据字典中抽取出来的。dbms_stats包的set_param存储过程可以用来设置默认值。要执行这个过程，需要analyze any dictionary和analyze any系统权限。dbms_stats包的get_param函数可以用来获取默认值。下面的例子展示了如何使用它们。注意，pname是参数的名称，而pval是参数的值：

```
SQL> execute dbms_output.put_line(dbms_stats.get_param(pname => 'CASCADE'))

DBMS_STATS.AUTO_CASCADE

SQL> execute dbms_stats.set_param(pname => 'CASCADE', pval =>'TRUE')

SQL> execute dbms_output.put_line(dbms_stats.get_param(pname => 'CASCADE'))

TRUE
```

另一个可以使用这种方法设置的参数是autostats_target。这个参数的唯一用途是，gather_stats_job任务可以使用该参数决定应该处理哪些对象的统计信息收集。表8-7列出了可选的值。其默认值是auto。

表8-7　autostats_target参数接受的值

值	含　义
all	处理所有的对象。直到11.2版本（包括在内），固定表都被排除在外。但是，从12.1版本开始，固定表都被包括在内
auto	由任务来决定应该处理哪些对象
oracle	只有属于数据字典的对象会被处理，固定表除外

要想无需多次执行get_param函数就获取所有参数的默认值，可以使用以下查询[①]：

```
SQL> SELECT sname AS parameter, nvl(spare4,sval1) AS default_value
  2  FROM sys.optstat_hist_control$
  3  WHERE sname IN ('CASCADE','ESTIMATE_PERCENT','DEGREE','METHOD_OPT',
  4                  'NO_INVALIDATE','GRANULARITY','AUTOSTATS_TARGET');

PARAMETER          DEFAULT_VALUE
```

① 遗憾的是，Oracle并不会通过一张数据字典视图显示这个信息，也就是说这个查询是基于内部表的。访问这张表需要select any dictionary系统权限。

```
---------------- ---------------------------
CASCADE          DBMS_STATS.AUTO_CASCADE
ESTIMATE_PERCENT DBMS_STATS.AUTO_SAMPLE_SIZE
DEGREE           NULL
METHOD_OPT       FOR ALL COLUMNS SIZE AUTO
NO_INVALIDATE    DBMS_STATS.AUTO_INVALIDATE
GRANULARITY      AUTO
AUTOSTATS_TARGET AUTO
```

要还原原来设置的默认值，可以使用dbms_stats包提供的reset_param_defaults存储过程。

8.4.2 现代方式

自11.1版本开始，为参数设置默认值的概念，被称作首选项，与10.2版本相比，该功能有了极大的增强。实际上，你不仅可以设置全局默认值，也可以在表级别设置默认值。这些增强的一个结果就是上一节中描述的get_param函数、set_param过程和reset_param_defaults过程都被淘汰了。

可以为参数autostats_target、cascade、concurrent、estimate_percent、degree、method_opt、no_invalidate、granularity、publish、incremental、stale_percent、table_cached_blocks（从11.2.0.4版本开始）以及从12.1版本开始出现的参数global_temp_table_stats、incremental_staleness和incremental_level等设置默认值。要修改它们，可以使用dbms_stats包提供的以下存储过程。

- set_global_prefs设置全局首选项。它取代了set_param存储过程。
- set_database_prefs设置数据库级首选项。全局首选项和数据库级别首选项的区别是后者不用作数据字典对象。换句话说，数据库级别首选项只作用于用户定义的对象。
- set_schema_prefs为某个特定的模式设置首选项。
- set_table_prefs为某个特定表设置首选项。

注意参数autostats_target和concurrent只能通过set_global_prefs存储过程来修改。

警告 存储过程set_database_prefs和set_schema_prefs不会直接将首选项信息存储到数据字典中，而是会将它们转变成为调用存储过程时即刻在数据库中或模式中可用的所有对象的表级别首选项。换句话说，真正存在的只有全局首选项或者表级别首选项。存储过程set_database_prefs和set_schema_prefs只是对存储过程set_table_prefs的简单包装。这意味着对于调用完这两个存储过程后创建的新表，将会使用全局首选项。

下面的PL/SQL代码块展示了如何为cascade参数设置不同的值。注意，pname指参数名称，pvalue指参数值，ownname指所有者，而tabname指表名。再强调一次，要非常小心，因为在这样的PL/SQL代码块中调用的顺序十分关键。实际上，每一次调用都会覆盖上一次调用完成的一些定义：

```
BEGIN
  dbms_stats.set_database_prefs(pname  => 'CASCADE',
                                pvalue => 'DBMS_STATS.AUTO_CASCADE');
  dbms_stats.set_schema_prefs(ownname => 'SCOTT',
                              pname   => 'CASCADE',
                              pvalue  => 'FALSE');
  dbms_stats.set_table_prefs(ownname => 'SCOTT',
```

```
                                    tabname => 'EMP',
                                    pname   => 'CASCADE',
                                    pvalue  => 'TRUE');
END;
```

为获取当前的设置，可以使用get_prefs函数来取代get_param函数。下面的查询用来展示在上面的PL/SQL块中所执行设置的效果。注意，pname是参数名称，ownname是所有者名称，而tabname是表名。正如你所看到的，依赖于所指定的参数，该函数会返回指定级别的值。这次对于首选项的搜索按照图8-9所示的方式进行：

```
SQL> SELECT dbms_stats.get_prefs(pname   => 'cascade') AS global,
  2         dbms_stats.get_prefs(pname   => 'cascade',
  3                              ownname => 'SCOTT',
  4                              tabname =>'DEPT') AS dept,
  5         dbms_stats.get_prefs(pname   => 'cascade',
  6                             ownname  => 'SCOTT',
  7                             tabname  =>'EMP') AS emp
  8 FROM dual;

GLOBAL                      DEPT  EMP
-----------------------     ----- ----
DBMS_STATS.AUTO_CASCADE     FALSE TRUE
```

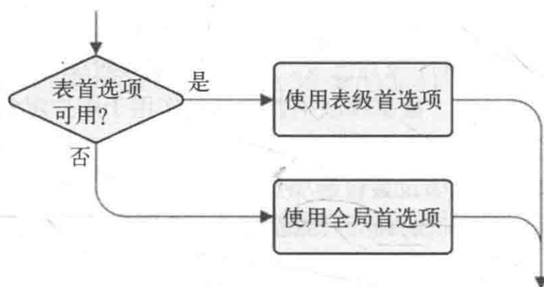


图8-9 在搜索首选项时，表级别设置优先于全局设置

如果希望不用多次执行get_param函数就可以获取多个全局首选项，正如上一节中描述的那样，可以查询内部数据字典表optstat_hist_control$。要获取表的首选项，也可以执行接下来的查询。注意，即便之前的PL/SQL代码块配置是在模式级别，dba_tab_stat_prefs视图仍显示了其设置结果：

```
SQL> SELECT table_name, preference_name, preference_value
  2 FROM dba_tab_stat_prefs
  3 WHERE owner = 'SCOTT'
  4 AND table_name IN ('EMP', 'DEPT')
  5 ORDER BY table_name, preference_name;

TABLE_NAME PREFERENCE_NAME PREFERENCE_VALUE
---------- --------------- ----------------
DEPT       CASCADE         FALSE
EMP        CASCADE         TRUE
```

要删除首选项，dbms_stats包提供了以下存储过程。

- reset_global_pref_defaults将全局首选项重置为默认值。

- delete_database_prefs在数据库级别删除首选项配置。
- delete_schema_prefs在模式级别删除首选项配置。
- delete_table_prefs在表级别删除首选项配置。

下面的调用展示了如何删除当前scott用户下包含的所有表中与cascade参数相关的首选项：

```
dbms_stats.delete_schema_prefs(ownname => 'SCOTT', pname => 'CASCADE')
```

要在全局级别和数据库级别执行这些存储过程，需要有analyze any dictionary和analyze any系统权限。要在模式级别或表级别执行这些存储过程，需要以所有者身份连接到数据库或者拥有analyze any系统权限。

8.5 处理全局临时表

直到11.2版本为止（包括11.2版本在内），对于全局临时表，dbms_stats包仅对gather_database_stats和gather_schema_stats存储过程提供gather_temp参数的支持。通过这个参数，仅能够控制是否处理全局临时表。收集的执行过程与“普通”表没有区别。结果，在大多数时间里，抛开对象统计信息是如何被收集的不说，全局临时表上没有可以使用的对象统计信息。原因有两个。第一，dbms_stats在处理过程开始会执行一个COMMIT操作，因此，通过on commit delete rows（也就是默认选项）选项创建的临时表永远是空的。第二，如果收集过程与往常一样，发生在一个像默认收集任务这样的任务中，全局临时表也是空的。总之，获取有意义的对象统计信息的唯一方式就是手工设置它们。但是，即使你手工设置了它们，也没有办法找到一组适合所有人的对象统计信息。实际上，每个会话都有可能在这些表中存储一组不同数量的数据。

最终，12.1版本引入了一个新特性来正确地处理全局临时表。其思路是你可以在共享统计信息（在11.2及之前的版本中唯一可用的选项）和会话统计信息之间进行选择。如果使用了会话统计信息（全局临时表的默认选项），每个会话都可以单独收集一组对其他会话并不可见的对象统计信息。收集的过程本身与往常一样，通过dbms_stats包的gather_table_stats存储过程来执行。这意味着要想从这个特性中获益，应用程序必须进行修改，以在全局临时表数据加载完毕后立刻执行对gather_table_stats过程的调用。注意，为了使这个特性发挥作用，dbms_stats包处理流程开始的COMMIT操作被移除了。下面的例子（来自于gtt.sql脚本）说明了这个特性是如何运作的：

```
SQL> CREATE GLOBAL TEMPORARY TABLE t (id NUMBER, pad VARCHAR2(1000));

SQL> INSERT INTO t SELECT rownum, rpad('*',1000,'*') FROM dual CONNECT BY level <= 1000;

SQL> execute dbms_stats.gather_table_stats(ownname => user, tabname => 't')

SQL> SELECT num_rows, blocks, avg_row_len, scope
  2  FROM user_tab_statistics
  3  WHERE table_name = 'T';

NUM_ROWS     BLOCKS AVG_ROW_LEN SCOPE
-------- ---------- ----------- -------
                                SHARED
    1000        147        1005 SESSION

SQL> SELECT count(*)
```

8

```
  2  FROM t
  3  WHERE id BETWEEN 10 AND 100;

  COUNT(*)
----------
        91

SQL> SELECT * FROM table(dbms_xplan.display_cursor);

-----------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |       |       |    42 (100)|          |
|   1 |  SORT AGGREGATE    |      |     1 |     4 |            |          |
|*  2 |   TABLE ACCESS FULL| T    |    92 |   368 |    42   (0)| 00:00:01 |
-----------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter(("ID"<=100 AND "ID">=10))
Note
-----
   - Global temporary table session private statistics used
```

要控制是使用共享统计信息还是使用会话统计信息，可以设置global_temp_table_stats首选项。受支持的值有两个：shared和session。默认值是session。

8.6　处理挂起的对象统计信息

通常，一旦收集过程结束，就会将对象统计信息发布到查询优化器（也就是说，使其可访问）。这意味着无法在不覆盖当前对象统计信息的情况下（例如，基于测试目的）收集统计信息。当然了，用于测试用途的应该是测试数据库，但有时候测试环境并不总是那么理想；你可能想在生产环境中做这样的测试。测试数据库中存储的数据与生产数据库中存储的数据不一致，就是这样的一个例子。

自11.1版本起，就可以将收集统计信息与发布它们的过程分隔开来，这样便可以使用未发布的对象统计信息，也就是所说的挂起的统计信息（pending statistics），将其用作测试用途。下面是处理过程（完整的例子在pending_object_statistics.sql脚本中提供）。

(1) 通过将publish首选项设置为FALSE来禁用自动发布（默认值是TRUE）。正如上一节所描述的，你可以通过全局、数据库、模式或者表级别来完成设置。下面的例子展示了如何为属于当前用户的一张表设置该首选项：

```
dbms_stats.set_table_prefs(ownname => user,
                           tabname => 'T',
                           pname   => 'PUBLISH',
                           pvalue  => 'FALSE')
```

(2) 收集对象统计信息。因为这张表的publish首选项被设置成FALSE，最近收集的对象统计信息没有被发布，而是创建了一组挂起的统计信息。这意味着查询优化器仍然使用此次收集之前可用的统计信息。同时，依赖于这张表的游标并没有失效：

```
dbms_stats.gather_table_stats(ownname => user, tabname => 'T')
```

(3) 要测试挂起的统计信息对一个应用程序或者一组SQL语句的影响，既可以通过在会话级别将初始化参数optimizer_use_pending_statistics设置为TRUE，也可以通过在SQL语句级别使用opt_param('optimizer_use_pending_statistics' 'true')这个hint。

(4) 如果测试成功，可以通过调用publish_pending_stats存储过程来发布挂起的统计信息（换句话说，使其对所有会话可用）。下面的例子会展示如何为单张表发布挂起的统计信息。如果将tabname参数设置为NULL，指定模式下所有挂起的统计信息都将被发布。这个过程还有两个额外的参数。如前所述，第三个参数no_invalidate控制依赖于修改的对象统计信息的游标是否失效。第四个参数force用于解开对象统计信息上潜在的锁（详见8.10节）。其默认值为FALSE，意思是对锁的处理遵守默认值：

```
dbms_stats.publish_pending_stats(ownname => user, tabname => 'T')
```

(5) 如果测试不成功，可以通过调用delete_pending_stats存储过程删除挂起的统计信息。如果没有指定tabname参数的值或将其设置为NULL，通过ownname参数指定的整个模式下挂起的统计信息都会被删除：

```
dbms_stats.delete_pending_stats(ownname => user, tabname => 'T')
```

(6) 通过将publish首选项设置为TRUE来启用自动发布。需要执行这一步来恢复第1步中执行的更改：

```
dbms_stats.set_table_prefs(ownname => user,
                           tabname => 'T',
                           pname   => 'PUBLISH',
                           pvalue  => 'TRUE')
```

要执行存储过程publish_pending_stats和delete_pending_stats，需要以所有者身份连接或者具有analyze any系统权限。

如果你有兴趣了解这些挂起的统计信息的值，下面的数据字典视图提供了所有必要的信息。对于每个视图，都有dba、all，以及在12.1多租户环境下的cdb版本。

- user_tab_pending_stats显示挂起的表统计信息。
- user_ind_pending_stats显示挂起的索引统计信息。
- user_col_pending_stats显示挂起的列统计信息。
- user_tab_histgrm_pending_stats显示挂起的直方图。

这些数据字典视图的结构分别类似于user_tab_statistics、user_ind_statistics、user_tab_col_statistics以及user_tab_histograms。

8.7 处理分区对象

为分区的表和索引收集对象统计信息是很有挑战性的。本节将具体描述这些挑战，并介绍两种应对这些挑战的技术。

8.7.1 挑战

dbms_stats包使用两种主要的方式为分区的表和索引收集对象统计信息。

- 在对象、分区以及子分区级别上通过在每个级别上分别执行的查询来收集对象统计信息。

- 只在物理层面（可能是分区级别也可能是子分区级别）收集对象统计信息，并使用其结果来推算出其他级别的对象统计信息。

下面是这两种方式的两个关键区别。

- 总的来说，第一种收集对象统计信息的方式需要的时间和资源要高很多。实际上，在表/索引级别收集对象统计信息时，必须要访问所有的段。同样的事情也会发生在子分区对象的分区级别。例如，一张包含多年数据的按星期分区的表。如果一个分区发生了变化，那么必须访问所有分区才能更新表/索引级别统计信息。甚至在只有一个分区的数据被修改了的情况下，也必须访问所有分区。
- 第二种方式消耗的资源则要少很多，但是这种方式只能在物理级别生成准确的统计信息。这是因为它无法从底层的分区和子分区推算出不重复值的数量和直方图。顺便说一下，其他所有的统计信息都可以推算出来。

通过第一种方式收集的对象统计信息叫作全局统计信息。通过第二种方式收集的统计信息叫作推算统计信息（有时也称作聚合统计信息）。要辨别收集的是哪种类型，可以检查表8-2中列举的数据字典视图中global_stats列值是YES还是NO。只要可能，dbms_stats包就会收集全局统计信息。dbms_stats包只会在某些情况下收集推算统计信息，例如，收集的粒度被明确地限制在子分区级别并且在分区以及表/索引级别没有可用的对象统计信息时。

接下来的例子基于脚本global_stats.sql生成的输出，展示了这样的案例：对于一张按范围分区并按照hash进行子分区的表，其推算统计信息并不准确。注意，不仅表和分区级别的不重复值数量有误，global_stats列也被设置为NO。

- 在一张没有对象统计信息的表上执行子分区级别的收集（注意，没有涉及采样）：

```
SQL> BEGIN
  2    dbms_stats.delete_table_stats(ownname => user,
  3                                  tabname => 't');
  4    dbms_stats.gather_table_stats(ownname          => user,
  5                                  tabname          => 't',
  6                                  estimate_percent => 100,
  6                                  granularity      => 'subpartition');
  7  END;
  8  /
```

- 在表级别的不重复值数量是错的，因为它们是通过推算统计信息收集的：

```
SQL> SELECT count(DISTINCT sp)
  2  FROM t;

 COUNT(DISTINCTSP)
--------------------
               100
SQL>
SQL> SELECT num_distinct, global_stats
  2  FROM user_tab_col_statistics
  3  WHERE table_name = 'T'
  4  AND column_name = 'SP';

NUM_DISTINCT GLOBAL_STATS
```

```
------------ ------------
          28 NO
```

- 在分区级别（这里指一个单独的分区）的不重复值数量也是错的，因为它们是通过推算统计信息收集的：

```
SQL> SELECT count(DISTINCT sp)
  2  FROM t PARTITION (q1);

  COUNT(DISTINCTSP)
--------------------
                 100

SQL>
SQL> SELECT num_distinct, global_stats
  2  FROM user_part_col_statistics
  3  WHERE table_name = 'T'
  4  AND partition_name = 'Q1'
  5  AND column_name = 'SP';

NUM_DISTINCT GLOBAL_STATS
------------ ------------
          28 NO
```

- 在子分区级别（这里是对于单个分区来说的）的不重复值数量是正确的：

```
SQL> SELECT 'Q1_SP1' AS part_name, count(DISTINCT sp) FROM t SUBPARTITION (q1_sp1)
  2  UNION ALL
  3  SELECT 'Q1_SP2', count(DISTINCT sp) FROM t SUBPARTITION (q1_sp2)
  4  UNION ALL
  5  SELECT 'Q1_SP3', count(DISTINCT sp) FROM t SUBPARTITION (q1_sp3)
  6  UNION ALL
  7  SELECT 'Q1_SP4', count(DISTINCT sp) FROM t SUBPARTITION (q1_sp4);

PART_NAME COUNT(DISTINCTSP)
--------- -----------------
Q1_SP1                   20
Q1_SP2                   28
Q1_SP3                   25
Q1_SP4                   27

SQL> SELECT subpartition_name, num_distinct, global_stats
  2  FROM user_subpart_col_statistics
  3  WHERE table_name = 'T'
  4  AND column_name = 'SP'
  5  AND subpartition_name LIKE 'Q1%';

SUBPARTITION_NAME NUM_DISTINCT GLOBAL_STATS
----------------- ------------ ------------
Q1_SP1                      20 YES
Q1_SP2                      28 YES
Q1_SP3                      25 YES
Q1_SP4                      27 YES
```

警告 表/分区级别的对象统计信息，只有当底层的所有分区都有合适的对象统计信息时，才可以从底层的分区推算出来。这也适用于从子分区统计信息推算分区统计信息。此外，要知道dbms_stats包不会使用推算统计信息替代全局统计信息。两种情形都可以通过脚本global_stats.sql重现出来。

概括起来，全局统计信息要比推算统计信息更加精确，但是需要更多的时间和资源来进行收集。有时候可能推算统计信息就足够了。因此在实践中，对于大表来说，在准确度与达到目的所需的时间和资源之间找到均衡点很重要。基于这个原因，接下来的两节将描述可以用来管理足够大的表的对象统计信息的技术，进而防止重复收集完全的全局统计信息。

8.7.2 增量统计信息

正如上一节中描述的那样，收集全局统计信息有优点也有缺点。主要的优点体现在表级别的对象统计信息的准确性上，如果使用了子分区，这个优点同样体现在分区级别。主要的缺点体现在收集它们所需要的资源和时间上。

增量统计信息的目标是在降低收集对象统计信息所需时间和资源的前提下提供相同的准确性。这怎么可能呢？其关键思路是在分区级别收集对象统计信息期间，利用存储在数据字典中的额外信息（称作概要信息），在表级别精确地推算对象统计信息。

要想从增量统计信息中获益必须首先满足以下要求。

- 正在运行的是11.1或之后的版本。
- 对于正在处理的表，其incremental首选项设置为TRUE（默认值是FALSE）：

```
dbms_stats.set_table_prefs(ownname => user,
                           tabname => 't',
                           pname   => 'incremental',
                           pvalue  => 'TRUE');
```

- 对于正在处理的表，其publish首选项设置为TRUE（默认值）。
- 对于正在处理的表，将参数estimate_percent设置为dbms_stats.auto_sample_size（默认值）。
- 在sysaux表空间中有可用剩余空间。

收集过程本身还是按照通常的方式进行，例如，通过对dbms_stats包的gather_table_stats存储过程的调用。唯一需要小心应对的是，要利用增量统计信息，必须在分区级别呈现概要信息。因此，设置完incremental首选项后，你必须在所有分区上收集新的对象统计信息。你可以认为在所有分区上收集新的对象统计信息的操作是最终启用增量统计信息的那个操作。也就是说，只满足上面列举的要求是不够的。

一旦所有的概要信息都就位了，dbms_stats包就会使用其监测信息来了解哪个分区（或子分区）被修改了，从而需要新的对象统计信息。因此，当使用增量统计信息时，不应该去瞄准被修改的分区（或子分区），而是应该让dbms_stats包自己找出它需要做的事情。下面的例子基于脚本incremental_stats.sql，就验证了这一点（仔细看一下last_analyzed时间戳来确定在哪些对象上收集了统计信息）：

```
SQL> SELECT object_type || ' ' || nvl(subpartition_name, partition_name) AS object,
  2         object_type, num_rows, blocks, avg_row_len,
  3         to_char(last_analyzed, 'HH24:MI:SS') AS last_analyzed
  4  FROM user_tab_statistics
  5  WHERE table_name = 'T'
  6  ORDER BY partition_name, subpartition_name;

OBJECT              OBJECT_TYPE  NUM_ROWS  BLOCKS AVG_ROW_LEN LAST_ANALYZED
------------------- ------------ -------- ------- ----------- -------------
SUBPARTITION Q1_SP1 SUBPARTITION     1786      46         116 14:52:22
SUBPARTITION Q1_SP2 SUBPARTITION     2173      46         116 14:52:22
PARTITION Q1        PARTITION        3959      92         116 14:52:22
SUBPARTITION Q2_SP1 SUBPARTITION     1804      46         116 14:52:22
SUBPARTITION Q2_SP2 SUBPARTITION     2200      46         116 14:52:22
PARTITION Q2        PARTITION        4004      92         116 14:52:22
SUBPARTITION Q3_SP1 SUBPARTITION     1815      46         116 14:52:22
SUBPARTITION Q3_SP2 SUBPARTITION     2233      46         116 14:52:22
PARTITION Q3        PARTITION        4048      92         116 14:52:22
SUBPARTITION Q4_SP1 SUBPARTITION     1795      46         116 14:52:22
SUBPARTITION Q4_SP2 SUBPARTITION     2194      46         116 14:52:22
PARTITION Q4        PARTITION        3989      92         116 14:52:22
TABLE               TABLE           16000     368         117 14:52:22

SQL> INSERT INTO t SELECT * FROM t SUBPARTITION (q1_sp1);

SQL> execute dbms_stats.gather_table_stats(ownname => user, tabname => 't', granularity=>'all')

SQL> SELECT object_type || ' ' || nvl(subpartition_name, partition_name) AS object,
  2         object_type, num_rows, blocks, avg_row_len,
  3         to_char(last_analyzed, 'HH24:MI:SS') AS last_analyzed
  4  FROM user_tab_statistics
  5  WHERE table_name = 'T'
  6  ORDER BY partition_name, subpartition_name;

OBJECT              OBJECT_TYPE  NUM_ROWS BLOCKS AVG_ROW_LEN LAST_ANALYZED
------------------- ------------ -------- ------ ----------- -------------
SUBPARTITION Q1_SP1 SUBPARTITION     3572    110         116 14:54:39
SUBPARTITION Q1_SP2 SUBPARTITION     2173     46         116 14:52:22
PARTITION Q1        PARTITION        5745    156         116 14:54:40
SUBPARTITION Q2_SP1 SUBPARTITION     1804     46         116 14:52:22
SUBPARTITION Q2_SP2 SUBPARTITION     2200     46         116 14:52:22
PARTITION Q2        PARTITION        4004     92         116 14:52:22
SUBPARTITION Q3_SP1 SUBPARTITION     1815     46         116 14:52:22
SUBPARTITION Q3_SP2 SUBPARTITION     2233     46         116 14:52:22
PARTITION Q3        PARTITION        4048     92         116 14:52:22
SUBPARTITION Q4_SP1 SUBPARTITION     1795     46         116 14:52:22
SUBPARTITION Q4_SP2 SUBPARTITION     2194     46         116 14:52:22
PARTITION Q4        PARTITION        3989     92         116 14:52:22
TABLE               TABLE           17786    432         117 14:54:40
```

如本例所示，与分区（或子分区）关联的对象统计信息在经历任何修改后都会被视为陈旧的。从12.1版本开始，有一个首选项incremental_staleness，你可以通过它控制这种行为。通过默认值NULL，这种行为与之前的版本表现一致（任何修改都会使一个分区变陈旧）。如果将值设置为

use_stale_percent，只有当修改的数量超过通过stale_percent首选项设置的阈值后，与分区（或子分区）关联的对象统计信息才会被认为是陈旧的。此外，通过值use_locked_stats，可以规定与拥有锁定的统计信息的分区（或子分区）关联的对象统计信息永不过期。注意可以同时启用use_stale_percent和use_locked_stats。下面是一个例子：

```
dbms_stats.set_table_prefs(ownname => user,
                           tabname => 't',
                           pname   => 'incremental_staleness',
                           pvalue  => 'use_stale_percent, use_locked_stats');
```

仅在12.1版本中，dbms_stats包可以在非分区表上创建概要信息（为此，必须在表上设置nincremental_leveln首选项）。结果，仅在12.1版本中，分区交换才可以利用增量统计信息。

提示　Oracle Support文档*How To Collect Statistics On Partitioned Table in 10g and 11g*（1417133.1）中提供了一个与增量统计信息有关的最重要的Bug和补丁列表。查看这篇文章来了解你正在运行的版本是否需要特别的关注，并进一步检查第一篇文章引用的其他文章。

8.7.3　复制统计信息

如果频繁地添加分区，并且它们的内容会随着时间发生显著的变化，在这样的情形中，保持一组有代表性的分区级别统计信息需要非常频繁的收集操作。这些频繁的收集操作代表着在资源使用方面的显著负载。此外，通常情况下，对最近添加的一个没有对象统计信息的分区不管不顾可不太好。这样做会导致动态采样的发生，这是第9章的一个特性。为了应对这样的问题，dbms_stats包通过copy_table_stats存储过程，提供从一个分区或子分区向另一个分区或子分区复制对象统计信息的功能。注意复制过程会像处理子分区和本地索引那样处理列统计信息和依赖的对象。

下面的命令演示了如何执行一个复制过程（完整案例见脚本copy_table_stats.sql）。ownname和tabname参数指定命令是在哪张表上执行。srcpartname和dstpartname参数分别指定源和目标分区（或子分区）：

```
dbms_stats.copy_table_stats(ownname      => user,
                            tabname      => 't',
                            srcpartname  => 'p_2014_q1',
                            dstpartname  => 'p_2015_q1',
                            scale_factor => 1);
```

有必要指出的是，copy_table_stats存储过程并非简单地执行一对一的复制。相反，它能够根据分区定义的方式来改变最大值和最小值。例如，对于一张给定的范围分区表，程序dbms_stats包能够从目标分区与之前分区的分区边界推算出其最小值和最大值。此外，自10.2.0.5版本开始，可以通过将scale_factor参数设置为1以外的值来按比例放大或缩小记录数和数据块数。

如果在执行复制的表上已经推算出表/索引级别的统计信息，那么在复制过程中表/索引级别的统计信息也会被修订。举例来说，记录数量增加了，也会相应地设置最大值。在子分区之间复制统计信息时，类似的事情也会发生在分区级别。

提示 直到11.2.0.3的所有版本都在copy_table_stats存储过程中包含一些bug。其中一些bug是你可能永远都不会遇到的稀有案例，但是另外的一些bug，根据你所运行的版本，可能会影响核心功能。在Oracle Support网站上搜索copy table stats来了解你正在使用的版本是否有需要特别注意的地方。

8.8 调度对象统计信息的收集

查询优化器需要对象统计信息来正确地完成它的使命。因此，创建新的数据库后，会默认设置一个调用dbms_stats包的gather_database_stats_job_proc存储过程的后台作业。gather_database_stats_job_proc存储过程执行的操作与调用dbms_stats包的gather_database_stats存储过程时使用选项参数gather_stale和gather_empty执行的操作在本质上是相同的。注意，虽然在10.2版本中使用的是正常的作业，但是从11.1版本开始起，会将收集过程集成在自动维护任务里面。在两种情况下，任务都是使用dbms_scheduler包调度的，而不是dbms_job包。

警告 在11.2及之前的版本中，默认情况下任务的目标是除了固定表以外的所有对象。因此，你必须自己在数据库引擎负载高峰期处理固定表的对象统计信息收集的工作。建议在负载高峰期收集数据是因为固定表的内容强烈依赖于负载。例如x$ksuse表，它为每个会话包含一条记录。

下面两个小节的主要目标是提供关于用于调度默认作业的配置的详细信息。其中第一节介绍10*g*版本。第二节介绍后续的版本。

8.8.1 10*g* 方式

gather_stats_job是在10*g*版本中自动设置的作业。其当前的配置，也就是下面示例中10.2版本的默认配置，可以通过下面的查询显示出来。输出是通过dbms_stats_job_10g.sql脚本生成的：

```
SQL> SELECT program_name, schedule_name, enabled, state
  2  FROM dba_scheduler_jobs
  3  WHERE owner = 'SYS'
  4  AND job_name = 'GATHER_STATS_JOB';

PROGRAM_NAME      SCHEDULE_NAME            ENABLED STATE
----------------- ------------------------ ------- ---------
GATHER_STATS_PROG MAINTENANCE_WINDOW_GROUP TRUE    SCHEDULED

SQL> SELECT program_action, number_of_arguments, enabled
  2  FROM dba_scheduler_programs
  3  WHERE owner = 'SYS'
  4  AND program_name = 'GATHER_STATS_PROG';

PROGRAM_ACTION                            NUMBER_OF_ARGUMENTS ENABLED
----------------------------------------- ------------------- -------
dbms_stats.gather_database_stats_job_proc                   0 TRUE
```

```
SQL> SELECT w.window_name, w.repeat_interval, w.duration, w.enabled
  2  FROM dba_scheduler_jobs j, dba_scheduler_wingroup_members m,
  3       dba_scheduler_windows w
  4  WHERE j.schedule_name = m.window_group_name
  5  AND m.window_name = w.window_name
  6  AND j.owner = 'SYS'
  7  AND j.job_name = 'GATHER_STATS_JOB';

WINDOW_NAME      REPEAT_INTERVAL                      DURATION      ENABLED
---------------- ------------------------------------ ------------- -------
WEEKNIGHT_WINDOW freq=daily;byday=MON,TUE,WED,THU,FRI; +000 08:00:00 TRUE
                 byhour=22;byminute=0; bysecond=0
WEEKEND_WINDOW   freq=daily;byday=SAT;byhour=0;byminut +002 00:00:00 TRUE
                 e=0;bysecond=0
```

总结起来，其配置如下。

- 作业执行gather_stats_prog程序并且可以在maintenance_window_group窗口组中运行。
- gather_stats_prog程序不使用任何参数调用dbms_stats包的gather_database_stats_job_proc存储过程。因为没有任何参数传递进来，唯一能够改变此存储过程行为的办法就是改变dbms_stats包的默认配置，正如8.4节中介绍的那样。注意这个存储过程是未公开的，并被标记为“仅供内部使用”。
- maintenance_window_group窗口组有两个成员：weeknight_window窗口和weekend_window窗口。前者从星期一到星期五每天晚上开放八个小时。后者在星期六和星期日开放。收集对象统计信息的任务在这两个窗口中的一个打开时执行。
- 作业、程序以及窗口都是启用的。

应该检查默认调度程序的开放时间和持续时长，并且在必要的时候，改变它们以精确匹配统计信息收集的频率。如有可能，它们应该匹配低负载的时间段。

每次作业因为窗口关闭而停止运行，都会产生一个包含所有没有来得及处理的对象列表的跟踪文件，并写入到由background_dump_dest初始化参数指定的目录中。下面是对这种跟踪文件的一段摘录：

```
GATHER_STATS_JOB: Stopped by Scheduler.
Consider increasing the maintenance window duration if this happens frequently.
The following objects/segments were not analyzed due to timeout:
TABLE: "SH"."SALES"."SALES_1995"
TABLE: "SH"."SALES"."SALES_1996"
TABLE: "SH"."SALES"."SALES_H1_1997"
...
TABLE: "SYS"."WRI$_OPTSTAT_AUX_HISTORY".""
TABLE: "SYS"."WRI$_ADV_OBJECTS".""
TABLE: "SYS"."WRI$_OPTSTAT_HISTGRM_HISTORY".""
error 1013 in job queue process
ORA-01013: user requested cancel of current operation
```

要启用或禁用gather_stats_job作业，可以使用下面的PL/SQL调用：

```
dbms_scheduler.enable(name => 'sys.gather_stats_job')

dbms_scheduler.disable(name => 'sys.gather_stats_job')
```

默认情况下，只有sys用户能够执行这些调用。其他用户需要alter object权限。举例来说，通过执行下面的SQL语句，system用户不仅能够修改而且也能删除gather_stats_job作业：

```
GRANT ALTER ON gather_stats_job TO system
```

8.8.2 11*g* 和 12*c* 方式

从11.1版本开始，对象统计信息的收集被集成到自动维护任务中。所以，上一节中描述的gather_stats_job作业就不复存在了。当前的配置，也就是下面例子中11.2版本的默认配置，可以通过下面的查询显示出来。输出部分是由dbms_stats_job_11g.sql脚本生成的：

```
SQL> SELECT task_name, status
  2  FROM dba_autotask_task
  3  WHERE client_name = 'auto optimizer stats collection';

TASK_NAME         STATUS
----------------- -------
gather_stats_prog ENABLED

SQL> SELECT program_action, number_of_arguments, enabled
  2  FROM dba_scheduler_programs
  3  WHERE owner = 'SYS'
  4  AND program_name = 'GATHER_STATS_PROG';

PROGRAM_ACTION                          NUMBER_OF_ARGUMENTS ENABLED
--------------------------------------- ------------------- -------
dbms_stats.gather_database_stats_job_proc                 0 TRUE

SQL> SELECT window_group
  2  FROM dba_autotask_client
  3  WHERE client_name = 'auto optimizer stats collection';

WINDOW_GROUP
--------------
ORA$AT_WGRP_OS

SQL> SELECT w.window_name, w.repeat_interval, w.duration, w.enabled
  2  FROM dba_autotask_window_clients c, dba_scheduler_windows w
  3  WHERE c.window_name = w.window_name
  4  AND c.optimizer_stats = 'ENABLED';

WINDOW_NAME      REPEAT_INTERVAL                                     DURATION      ENABLED
---------------- --------------------------------------------------- ------------- -------
MONDAY_WINDOW    freq=daily;byday=MON;byhour=22;byminute=0; bysecond=0 +000 04:00:00 TRUE
TUESDAY_WINDOW   freq=daily;byday=TUE;byhour=22;byminute=0; bysecond=0 +000 04:00:00 TRUE
WEDNESDAY_WINDOW freq=daily;byday=WED;byhour=22;byminute=0; bysecond=0 +000 04:00:00 TRUE
THURSDAY_WINDOW  freq=daily;byday=THU;byhour=22;byminute=0; bysecond=0 +000 04:00:00 TRUE
FRIDAY_WINDOW    freq=daily;byday=FRI;byhour=22;byminute=0; bysecond=0 +000 04:00:00 TRUE
SATURDAY_WINDOW  freq=daily;byday=SAT;byhour=6;byminute=0; bysecond=0  +000 20:00:00 TRUE
SUNDAY_WINDOW    freq=daily;byday=SUN;byhour=6;byminute=0; bysecond=0  +000 20:00:00 TRUE
```

总结起来，其配置如下。

- gather_stats_prog程序不使用任何参数调用dbms_stats包的gather_database_stats_job_proc过程。因为没有任何参数传递进来，唯一能够改变此过程的行为的办法就是改变dbms_stats包的默认配置，如8.4节所述。注意这个过程是未公开的，并被标记为“仅供内部使用”。
- 用于自动维护任务的窗口组有七个成员，一个星期中的每一天对应一个。从星期一到星期五，每天开放四个小时。从星球六到星期日，每天开放20个小时。收集对象统计信息的任务会在这些窗口中的一个打开时执行。注意当一个窗口打开了很长时间后，例如在周末，gather_stats_prog程序每隔四个小时重启一次。
- 维护任务、程序以及窗口都是启用的。

应检查默认调度程序的开放时间和持续时长，并且在必要的时候，改变它们以精确匹配统计信息收集的频率。如有可能，它们应该匹配低负载的时间段。

要完全启用或禁用维护任务，可以使用下面的PL/SQL调用：通过将windows_name参数设置为一个非空值，还可以为一个单独的窗口启用或禁用维护任务。

```
dbms_auto_task_admin.enable(client_name => 'auto optimizer stats collection',
                            operation   => NULL,
                            window_name => NULL)

dbms_auto_task_admin.disable(client_name => 'auto optimizer stats collection',
                             operation   => NULL,
                             window_name => NULL)
```

警告　从11.2版本开始，将job_queue_processes初始化参数设置为0即可禁用自动统计信息作业（以及其他所有通过该Scheduler调度的任务）。

8.9　还原对象统计信息

无论何时通过dbms_stats包收集了对象统计信息，或者从11.2版本开始，用ALTER INDEX语句取代简单地使用新的统计信息覆盖当前统计信息，当前统计信息都会被存储到其他数据字典表中，并保存一份在保留期内出现变化的所有历史记录。其用途是，万一新的统计信息导致了效率低下的执行计划，可以还原旧的统计信息。

对象统计信息在历史中保存一段由保留期指定的时间间隔（系统统计信息也是这样，因为它们是由相同的基础功能维护的）。默认值是31天。可以通过调用dbms_stats包的get_stats_history_retention函数来显示当前值，如下所示：

```
SELECT dbms_stats.get_stats_history_retention() AS retention FROM dual
```

要修改保留期，可以使用dbms_stats包提供的alter_stats_history_retention存储过程。下面是一个将保留期设置为14天的调用例子：

```
dbms_stats.alter_stats_history_retention(retention => 14)
```

注意，使用alter_stats_history_retention存储过程时，下面的值有特殊意义。

- NULL会将保留期设置为默认值。

- 0会禁用历史记录。
- -1会禁用历史记录的清除。

将statistics_level初始化参数设置为typical（默认值）或者all时，时间超出保留期的统计信息会被自动清除掉。一旦有必要进行手工清除时，可以使用dbms_stats包提供的purge_stats存储过程。下面的调用清除历史记录中所有超过14天的统计信息：

```
dbms_stats.purge_stats(before_timestamp => systimestamp - INTERVAL '14' DAY)
```

要执行alter_stats_history_retention和purge_stats存储过程，需要有analyze any和analyze any dictionary系统权限。

如果想知道对于一张给定的表它的对象统计信息何时被修改过，user_tab_stats_history数据字典视图可以提供所有必要的信息。当然了，还有dba、all以及12.1版本中多租户环境下的cdb版本可用。下面是一个例子。通过下面的查询，可以显示sys模式下的tab$表的对象统计信息的修改时间：

```
SQL> SELECT stats_update_time
  2  FROM dba_tab_stats_history
  3  WHERE owner = 'SYS' and table_name = 'TAB$';

STATS_UPDATE_TIME
--------------------------------
26-MAR-14 22.03.03.104730 +01:00
27-MAR-14 22.01.14.193033 +01:00
13-APR-14 14.14.57.461660 +02:00
```

无论什么时候，如果有必要，都可以从历史记录中还原统计信息。出于这个目的，dbms_stats提供以下存储过程。

- restore_database_stats为整个数据库还原对象统计信息。
- restore_dictionary_stats为数据字典还原对象统计信息。
- restore_fixed_objects_stats为固定表及其索引还原对象统计信息。
- restore_schema_stats为单个模式还原对象统计信息。
- restore_table_stats为单张表还原对象统计信息。

除了指定目标的参数之外（例如，restore_table_stats过程的模式和表名），所有这些存储过程都提供以下参数。

- as_of_timestamp指定将统计信息还原至某一特定的时间点。
- force指定是否可以覆盖锁定的统计信息。注意统计信息上的锁也是历史记录的一部分。这就意味着关于统计信息的状态信息（锁定与否）也可以被还原。默认值为FALSE。
- no_invalidate指定依赖于被覆盖的统计信息的游标是否失效。这个参数接受的值为TRUE、FALSE，还有dbms_stats.auto_invalidate。默认值是dbms_stats.auto_invalidate。

下面的调用将SH模式下的对象统计信息还原为一天以前使用的值。因此，force参数被设置为TRUE时，即使当前统计信息是锁定状态也会被还原：

```
dbms_stats.restore_schema_stats(ownname         => 'SH',
                                as_of_timestamp => systimestamp - INTERVAL '1' DAY,
                                force           => TRUE)
```

8.10 锁定对象统计信息

在某些情况下，可能需要确保数据库的部分对象统计信息不可用或者不允许修改，这是因为需要使用动态采样（见第9章），或者必须使用非最新的对象统计信息（例如，因为某些表的内容变化非常频繁，你希望只有在这些表包含了一组有代表性的数据时才小心地收集其状态），也可能因为收集统计信息不可行（例如，出现了bug）。

可以通过执行下面的dbms_stats包中的存储过程来显式锁定对象统计信息。注意这些锁和通常所说的数据库锁没有任何关系。实际上，它们是在数据字典的表级别设置的简单标记。

- lock_schema_stats锁定属于某个模式下的所有表的对象统计信息：

```
dbms_stats.lock_schema_stats(ownname => user)
```

- lock_table_stats锁定单张表的对象统计信息：

```
dbms_stats.lock_table_stats(ownname => user, tabname => 'T')
```

当然，也可以移除这些锁，你可以通过以下存储过程中的一个来完成。

- unlock_schema_stats解除某个模式下所有表的对象统计信息上的锁定：

```
dbms_stats.unlock_schema_stats(ownname => user)
```

- unlock_table_stats解除单张表的对象统计信息上的锁定：

```
dbms_stats.unlock_table_stats(ownname => user, tabname => 'T')
```

要执行这四个存储过程，需要以所有者身份登录或者拥有analyze any系统权限。

锁定了某张表的对象统计信息时，会将与该表相关的所有对象统计信息（包括表统计信息、列统计信息、直方图以及所有依赖索引的索引统计信息）都视为锁定的。

锁定了某张表的对象统计信息的情况下，dbms_stats包中修改单张表的对象统计信息的过程（例如gather_table_stats）会引发一个错误（ORA-20005）。与此相反，操作多张表的过程（例如gather_schema_stats）会跳过锁定的表。大多数修改对象统计信息的过程能够通过将force参数设置为TRUE来覆盖锁定。下面的例子演示了这种行为（完整示例参见lock_statistics.sql脚本）：

```
SQL> BEGIN
  2    dbms_stats.lock_schema_stats(ownname => user);
  3  END;
  4  /

SQL> BEGIN
  2    dbms_stats.gather_schema_stats(ownname => user);
  3  END;
  4  /

SQL> BEGIN
  2    dbms_stats.gather_table_stats(ownname => user,
  3                                  tabname => 'T');
  4  END;
  5  /
BEGIN
*
ERROR at line 1:
```

```
ORA-20005: object statistics are locked (stattype = ALL)
ORA-06512: at "SYS.DBMS_STATS", line 33859
ORA-06512: at line 2

SQL> BEGIN
  2    dbms_stats.gather_table_stats(ownname => user,
  3                                  tabname => 'T',
  4                                  force   => TRUE);
  5  END;
  6  /
```

要想知道哪些表的对象统计信息被锁定了，可以使用类似下面这样的查询：

```
SQL> SELECT table_name
  2  FROM user_tab_statistics
  3  WHERE stattype_locked IS NOT NULL;

TABLE_NAME
----------
T
```

要知道并非只有dbms_stats包会收集对象统计信息，因此，也不是只有它才会受对象统计信息上的锁影响。实际上，ANALYZE、CREATE INDEX和ALTER INDEX语句，以及12.1及之后版本的CTAS语句和向空表执行直接路径插入，也都会收集对象统计信息。ANALYZE语句会在被明确告知时收集对象统计信息。但是，如本章开头所述，你不应该再使用这个语句收集统计信息。其余的语句会在执行它们分配的任务时自动收集对象统计信息。这样做很有意义，因为执行这些SQL语句时收集统计信息的开销可以忽略不计。所以，锁定了表的对象统计信息时，以上这些SQL语句的行为可能会有所不同或者甚至会失败。下面的例子作为上一个例子的延续，展示了这种行为：

```
SQL> ANALYZE TABLE t COMPUTE STATISTICS;
ANALYZE TABLE t COMPUTE STATISTICS
*
ERROR at line 1:
ORA-38029: object statistics are locked

SQL> ANALYZE TABLE t VALIDATE STRUCTURE;

SQL> ALTER INDEX t_pk REBUILD COMPUTE STATISTICS;
ALTER INDEX t_pk REBUILD COMPUTE STATISTICS
*
ERROR at line 1:
ORA-38029: object statistics are locked

SQL> ALTER INDEX t_pk REBUILD;

SQL> CREATE INDEX t_i ON t (pad) COMPUTE STATISTICS;
CREATE INDEX t_i ON t (pad) COMPUTE STATISTICS
                   *
ERROR at line 1:
ORA-38029: object statistics are locked

SQL> CREATE INDEX t_i ON t (pad);
```

注意，SQL语句CREATE INDEX和ALTER INDEX只有在指定了不推荐使用的COMPUTE STATISTICS子句时才会失败。因为这些SQL语句都会默认收集对象统计信息，使用COMPUTE STATISTICS子句完全没有意义。

8.11 比较对象统计信息

在下面三种常见情形中，你最终会为同一个对象生成多组对象统计信息。

- 当你命令dbms_stats包（通过参数statown、stattab和statid）将当前对象统计信息保存到备份表中时。
- dbms_stats包被用于收集对象统计信息时。事实上，如8.9节所述，当收集一组新的统计信息时，程序包会自动保存对象统计信息的历史记录而不是简单地对其进行覆盖。
- 从11.1版本开始，当你收集挂起的统计信息时。

通常情况下你希望了解两组对象统计信息之间的不同。自10.2.0.4版本开始，你不再需要自己动手写查询语句完成这样的比较。可以简单地利用dbms_stats包提供的新函数。

下面的例子基于comparing_object_statistics.sql脚本输出，显示了此类报告：

```
###############################################################################

STATISTICS DIFFERENCE REPORT FOR:
.................................

TABLE         : T
OWNER         : CHRIS
SOURCE A      : Statistics as of 10-APR-13 20.05.07.106712 +02:00
SOURCE B      : Current Statistics in dictionary
PCTTHRESHOLD : 10
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
TABLE / (SUB)PARTITION STATISTICS DIFFERENCE:
.............................................

OBJECTNAME                  TYP SRC ROWS       BLOCKS      ROWLEN       SAMPSIZE
...............................................................................
T                           T   A   10088      110         37           5865
                                B   12691      253         37           5036
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
COLUMN STATISTICS DIFFERENCE:
.............................

COLUMN_NAME     SRC NDV     DENSITY     HIST NULLS    LEN   MIN   MAX   SAMPSIZ
...............................................................................
ID              A   9862    .000101399  NO   0        4     C103  C2646 5734
                B   12645   .000079082  NO   0        5     C108  C3026 5018
VAL1            A   3203    .000454959  YES  0        5     3D382 C2240 5779
                B   2990    .000489236  YES  0        5     3D421 C2251 4926
VAL2            A   9       .000049759  YES  0        3     C10C  C114  5842
                B   9       .000039438  YES  0        3     C10C  C114  5031
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
INDEX / (SUB)PARTITION STATISTICS DIFFERENCE:
```

```
...........................................

OBJECTNAME      TYP SRC ROWS    LEAFBLK DISTKEY LF/KY DB/KY CLF     LVL SAMPSIZ
...............................................................................

                                 INDEX: T_PK
                                 ...........

T_PK             I   A   10000   20      10000   1     1     9901    1   10000
                     B   12500   27      12500   1     1     12300   1   12500
###############################################################################
```

注意开头部分的内容，你可以看到用于比较的参数：模式和表的名称、两个比较源的定义（A和B）以及一个阈值。最后的这个参数指定是否只显示两组统计信息之间的差异（按百分比）达到指定阈值的那些对象统计信息。例如，如果你有两个值100和115，仅当阈值设置为15或者更低的时候它们才会被认为是不同的。默认的阈值是10。要显示所有的对象统计信息，可以使用值0。

下面是dbms_stats包中可以用来生成这样的报告的函数。

- diff_table_stats_in_stattab用于比较一张备份表（通过参数ownname和tabname指定）中的对象统计信息与当前对象统计信息之间的差异，或者比较其与另外一张备份表中的另一组信息之间的差异。参数stattab1、statid1以及stattab1own用来指定第一张备份表。第二张备份表（此处是可选的）是通过参数stattab2、statid2以及 stattab2own指定的。如果第二张备份表的参数没有指定，或者它们被设置为NULL，那么当前对象统计信息就会与第一张备份表的对象统计信息进行比较。下面的例子将当前对象统计信息与一组存储在mystats备份表中、名为set1的对象统计信息进行比较：

```
dbms_stats.diff_table_stats_in_stattab(ownname      => user,
                                       tabname      => 'T',
                                       stattab1     => 'MYSTATS',
                                       statid1      => 'SET1',
                                       stattab1own  => user,
                                       pctthreshold => 10)
```

- diff_table_stats_in_history比较一张表的当前对象统计信息与历史记录中的对象统计信息，或者比较这张表历史记录中的两组对象统计信息。参数time1和time2用来指定使用哪些对象统计信息比较。如果time2没有指定，或者设置为NULL，则当前对象统计信息与历史记录中的另一组进行比较。下面的例子将当前对象统计信息与一天之前的对象统计信息（例如，在夜间执行的统计信息收集之前的对象统计信息）进行比较：

```
dbms_stats.diff_table_stats_in_history(ownname      => user,
                                       tabname      => 'T',
                                       time1        => systimestamp - 1,
                                       time2        => NULL,
                                       pctthreshold => 10));
```

- diff_table_stats_in_pending将一张表的当前对象统计信息或者历史记录中的一组信息，与挂起的统计信息进行比较。要想指定存储在历史记录中的对象统计信息，可以使用参数time_stamp。如果这个参数设置为NULL（默认值），则当前对象统计信息与挂起的统计信息进行比较。下面的例子将当前的统计信息与挂起的统计信息进行比较：

```
dbms_stats.diff_table_stats_in_pending(ownname      => user,
                                       tabname      => 'T',
                                       time_stamp   => NULL,
                                       pctthreshold => 10));
```

8.12　删除对象统计信息

可以从数据字典中删除对象统计信息。除非是测试需要，否则通常没有必要这样做。虽然如此，但也可能出现你想利用动态采样（参见第9章）而不希望某张表上有对象统计信息的情况。如果那样的话，就可以使用dbms_stats包中的下列过程：delete_database_stats、delete_dictionary_stats、delete_fixed_objects_stats、delete_schema_stats、delete_table_stats、delete_column_stats以及delete_index_stats。

正如你所看到的，对于每个gather_*_stats过程，都有一个对应的delete_*_stats过程。前者收集对象统计信息，后者删除对象统计信息。唯一的例外是delete_column_stats过程。顾名思义，它用来删除列统计信息和直方图。

表8-8总结了这些过程中的每一个可以使用的参数。大部分参数是相同的，因而它们与gather_*_stats过程中使用的参数具有相同的含义。这里只描述一些在之前的过程中尚未涉及的参数。

- cascade_parts指定是否删除所有底层分区的统计信息。这个参数接受的值为TRUE和FALSE。默认值为TRUE。
- cascade_columns指定是否同时删除列统计信息。这个参数接受的值为TRUE和FALSE。默认值为TRUE。
- cascade_indexes指定是否同时删除索引统计信息。这个参数接受的值为TRUE和FALSE。默认值为TRUE。
- col_stat_type指定删除哪些统计信息。如果将它设置为ALL，则删除列统计信息和直方图。如果将它设置为HISTOGRAM，则只删除直方图。默认值是NULL。这个参数从11.1版本开始可用。
- stat_category指定删除哪种类别的统计信息。它接受以逗号分隔的列表形式的值。如果指定了OBJECT_STATS，则对象统计信息（表统计信息、列统计信息、直方图以及索引统计信息）会被删除。如果指定了SYNOPSES，则只有支持增量统计信息的信息会被删除。默认情况下，对象统计信息和概要信息都会被删除。这个参数从12.1版本开始可用。

表8-8　用于删除对象统计信息的过程的参数

参　数	数据库	数据字典	固定表	模　式	表	列	索引
目标对象							
ownname				✓	✓	✓	✓
indname							✓
tabname					✓	✓	
colname						✓	
partname					✓	✓	✓
cascade_parts					✓	✓	✓
cascade_columns					✓		

（续）

参　数	数据库	数据字典	固定表	模　式	表	列	索引
cascade_indexes					✓		
stat_category	✓	✓		✓	✓		
col_stat_type						✓	
force	✓	✓	✓	✓	✓	✓	✓
删除选项							
no_invalidate	✓	✓	✓	✓	✓	✓	✓
备份表							
stattab	✓	✓	✓	✓	✓	✓	✓
statid	✓	✓	✓	✓	✓	✓	✓
statown	✓	✓	✓	✓	✓	✓	✓

下面的调用展示了如何在不修改其他统计信息的情况下删除一个列的直方图（完整示例参见delete_histogram.sql）：

```
dbms_stats.delete_column_stats(ownname       => user,
                               tabname       => 'T',
                               colname       => 'VAL',
                               col_stat_type => 'HISTOGRAM')
```

8.13 导出、导入、获取和设置对象统计信息

如图8-1所示，除了用于收集统计信息的过程和函数之外，dbms_stats包还提供了其他几个可用的过程和函数。在这里不会介绍它们，因为它们在实践中很少使用。相关信息请查看*Oracle Database PL/SQL Packages and Types Reference*手册。尽管如此，我还是想分享一下下面这个你在文档中找不到的小知识。

警告 由dbms_stats包提供的导出和导入过程使用8.10节中描述的技术来处理锁定问题。唯一的例外，仅会出现在11.2.0.2之前的版本中，与没有对象统计信息的表有关。对于这样的表，当它们的对象统计信息被导出或者导入后，其统计信息的锁就会消失。

8.14 管理操作的日志记录

dbms_stats包中的许多过程在数据字典中记录关于它们的执行的信息。这些日志信息通过dba_optstat_operations以及在12.1多租户环境下的cdb_optstat_operations视图来予以展现。基本上，你可以查到执行了哪些操作，它们是什么时候开始执行的以及执行了多久。从12.1版本开始，关于状态[①]、会话以及与操作相关的作业信息（可选）都可以访问。接下来的例子摘自一个生产数据库，显示了

① 可能出现的值有：PENDING、IN PROGRESS、COMPLETED、FAILED、SKIPPED以及TIMED OUT。

gather_database_stats过程每天都会启动并花费9~18分钟来执行（注意，2014年4月5、6日是周末）：

```
SQL> SELECT operation, start_time,
  2         (end_time-start_time) DAY(1) TO SECOND(0) AS duration
  3  FROM dba_optstat_operations
  4  ORDER BY start_time DESC;

OPERATION                  START_TIME                          DURATION
-------------------------- ----------------------------------- -----------
gather_database_stats(auto) 09-APR-14 10.00.08.877925 PM +02:00 +0 00:10:28
gather_database_stats(auto) 08-APR-14 10.00.02.899209 PM +02:00 +0 00:09:30
gather_database_stats(auto) 07-APR-14 10.00.04.119250 PM +02:00 +0 00:12:45
gather_database_stats(auto) 06-APR-14 10.05.00.173419 PM +02:00 +0 00:00:55
gather_database_stats(auto) 06-APR-14 06.04.46.957190 PM +02:00 +0 00:00:50
gather_database_stats(auto) 06-APR-14 02.04.32.438573 PM +02:00 +0 00:00:53
gather_database_stats(auto) 06-APR-14 10.04.16.208319 AM +02:00 +0 00:01:27
gather_database_stats(auto) 06-APR-14 06.00.09.299059 AM +02:00 +0 00:04:55
gather_database_stats(auto) 05-APR-14 10.03.28.888807 PM +02:00 +0 00:00:58
gather_database_stats(auto) 05-APR-14 06.03.14.637546 PM +02:00 +0 00:00:43
gather_database_stats(auto) 05-APR-14 02.02.59.997594 PM +02:00 +0 00:01:06
gather_database_stats(auto) 05-APR-14 10.02.46.052860 AM +02:00 +0 00:01:13
gather_database_stats(auto) 05-APR-14 06.00.03.801439 AM +02:00 +0 00:06:05
gather_database_stats(auto) 04-APR-14 10.00.03.068541 PM +02:00 +0 00:17:32
gather_database_stats(auto) 03-APR-14 10.00.02.781440 PM +02:00 +0 00:06:59
gather_database_stats(auto) 02-APR-14 10.00.02.702294 PM +02:00 +0 00:12:45
gather_database_stats(auto) 01-APR-14 10.00.03.254860 PM +02:00 +0 00:12:48
```

此外，从12.1版本开始，你可以查询到某个操作执行时使用的参数。例如，下面的查询展示了默认收集作业的最后一次执行都使用了哪些参数：

```
SQL> SELECT x.*
  2  FROM dba_optstat_operations o,
  3       XMLTable('/params/param'
  4                PASSING XMLType(notes)
  5                COLUMNS name VARCHAR2(20) PATH '@name',
  6                        value VARCHAR2(30) PATH '@val') x
  7  WHERE operation = 'gather_database_stats (auto)'
  8  AND start_time = (SELECT max(start_time)
  9                    FROM dba_optstat_operations
 10                    WHERE operation = 'gather_database_stats (auto)');

NAME                 VALUE
-------------------- ------------------------------
block_sample         FALSE
cascade              NULL
concurrent           FALSE
degree               NULL
estimate_percent     DEFAULT_ESTIMATE_PERCENT
granularity          DEFAULT_GRANULARITY
method_opt           DEFAULT_METHOD_OPT
no_invalidate        DBMS_STATS.AUTO_INVALIDATE
reporting_mode       FALSE
stattype             DATA
```

要知道日志信息是使用与之前描述的统计信息历史相同的机制来清除的。因此，两者拥有相同的保留期。

8.15 保持对象统计信息为最新的策略

dbms_stats包为管理对象统计信息提供了很多特性。问题是，应该如何以及何时使用它们来实现一个成功的配置？这个问题很难回答。恐怕不存在绝对的答案。换句话说，没有一个单独的方法能够适用于所有的情况。我们来研究一下如何处理这个问题。

基本的原则，也可能是最重要的一条，就是查询优化器需要对象统计信息来描述存储在数据库中的数据。因此当数据变化时，对象统计信息也应该跟着变化。你可能也清楚，我是提倡定期收集对象统计信息的。那些反对这项实践的人会争论说，如果一个数据库运行良好，就没有必要重新收集对象统计信息。这种方法的问题通常是，一些对象统计信息依赖于真实的数据。例如，有这样一种统计信息，其经常变化的是那些包含数据（比如与某个交易、某个销售或某个电话相关联的时间戳）的列的低/高值。诚然，在一般的表中它们的变化占少数，但是通常变化的那部分很关键，因为它们会在应用程序中被反复使用。在实践中，我遇到过的由于没有最新的对象统计信息而导致的问题比反过来的情况要多得多。

显而易见，在永远不变的数据上收集对象统计信息毫无意义。只有陈旧的对象统计信息应该被重新收集。因此，利用记录出现在每张表上的修改数量的特性就显得必不可少。用这种方法，你可以只对那些经历了大量修改的表重新收集统计信息。默认情况下，当一张表有超过10%的数据发生了变化就被认为是陈旧的。这是个合理的默认值。从11.1版本开始，如果有必要可以修改这个默认值。

收集对象统计信息的频率也是一个存在不同看法的问题。我见过各种成功的案例，有按小时的，有按月的，甚至有更低频率的。这其实依赖于你的数据。无论如何，当使用表的陈旧属性作为重新收集对象统计信息的基础时，太长的间隔会导致过量的陈旧对象出现，而太短的间隔又会导致统计信息收集需要过多的时间以及资源使用出现高峰。为此，我喜欢将它们安排得更频繁（为了分散负载）并保持单一的运行时间尽可能地短。如果你的系统有每天或每周的低使用率时段，那么将收集安排在那些时间段通常是一个好主意。如果你的系统是一个真正的7 × 24系统，那通常更好的做法是尽可能使用非常频繁的调度（每天执行许多次）来分散负载并避开高峰期。

如果你有加载或修改大量数据的作业（例如，在数据仓库环境中的ETL作业），就不应该等候一个已安排的对象统计信息的收集完成。而要直接将要修改的对象的统计信息收集作为作业本身的一部分。换句话说，如果你知道某些事物发生了大量变化，应立即触发统计信息的收集。

如果出于某个原因，你觉得不应该在某些表上收集对象统计信息，那就锁定它们。这样的话，定期收集对象统计信息的作业就会直接跳过这些表。这比完全禁止整个数据库的作业活动要好得多。

应该尽可能多地利用默认的收集作业。要在这方面满足你的要求，你应该检查默认的配置，如果有必要则进行更改。因为在对象级别的配置仅从11.1版本开始才可用，如果在之前的版本中对某些表有特别的需求，应该在默认作业之前安排一个作业来处理它们。通过这种方式，只会处理拥有陈旧统计信息对象的默认作业，而会直接跳过已处理的表。锁定可能也会有助于确保仅特定作业才会在那些关键表上重新收集对象统计信息。

相反，如果你正在考虑完全禁止默认的收集作业，则应该为oracle设置autostats_target首选项。

那样的话，就让数据库引擎处理好数据字典，而对于其他的表，可以设置一个具体的作业来完成你所期望的工作。

如果收集的统计信息导致无效率的执行计划，那么你可以做两件事。第一是通过还原本次收集统计信息之前顺利使用的对象统计信息来修复问题。第二是找出为什么查询优化器使用新的对象统计信息会生成无效率的执行计划。为此，你首先应该检查最近收集的统计信息是否正确地描述了数据。举例来说，有可能伴随着新的数据分布的采样会导致不同的直方图。如果对象统计信息不良，那么收集本身，或者收集使用的参数就是问题所在。如果实际上对象统计信息没有问题，那么还有两种可能的原因。要么是查询优化器没有正确配置，要么是查询优化器犯了错误。你几乎无法控制后者，但是应该能够为前者找到解决方案。无论如何，应该避免匆忙认定是收集对象统计信息的固有问题，而因此停止定期收集它们。

最佳实践是使用dbms_stats包收集对象统计信息。但是，确实存在正确的对象统计信息误导查询优化器的情况。一个常见的例子是历史数据必须保持在线很长时间（举个例子，在瑞士某些类型的数据必须至少保存十年）。在这种情况下，如果数据分布几乎不随着时间发生改变，通过dbms_stats包收集的对象统计信息应该没有问题。与此相反，如果数据分布严重依赖于时间段而且应用程序经常只访问数据的一部分，那么就有理由手工修改（也就是，捏造）对象统计信息用以描述大部分相关数据。换句话说，如果你知道dbms_stats包忽略的或者无法发现的某些东西，那么就可以合理使用捏造的对象统计信息来告知查询优化器。

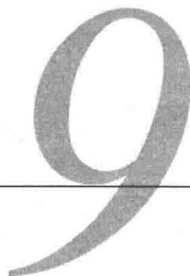
8.16　小结

本章描述了什么是表统计信息、列统计信息、直方图和索引统计信息，并依次说明它们是如何描述存储在数据库中的数据的。本章还涵盖了如何使用dbms_stats包收集对象统计信息以及在数据字典中从哪里找到它们。

本章没有详细描述对象统计信息的使用。相关内容以及配置查询优化器的初始化参数信息会在下一章进行介绍。学习完第9章，你应该能够正确地配置查询优化器，并且会在大部分时间里得到高效的执行计划。

第 9 章

配置查询优化器



查询优化器对SQL语句的性能负有直接责任。基于这个原因，我们有必要花一些时间将它配置好。事实上，并不存在一个最优化的配置，查询优化器可能会产生低效率的执行计划，从而导致不理想的性能表现。

查询优化器的配置不仅仅由几个初始化参数组成，还包括系统统计信息和对象统计信息（参见第7章、第8章）。本章将描述这些初始化参数和统计信息如何影响查询优化器，并展示一个简单实用的路线图以帮助你获得合理的配置。

警告 除了一个公式之外，Oracle没有公布本章所提供的其他公式。一些测试表明这些公式能够描述查询优化器是如何估算一个给定操作的成本的。但无论如何，也不能说在所有情形中它们都是精确的或者正确的。之所以提供它们是为了给你一个思路，让你了解初始化参数或统计信息是如何影响查询优化器估算的。

9.1 配置还是不配置

鉴于我们的情况，正像一句肯尼亚谚语[1]说的那样，我会说："配置查询优化器代价高昂，但是却值得。"在实践中，我见过太多低估了一个良好配置的重要性的站点。有时我甚至会与那些对我说出下面这样的话的人进行激烈讨论："我们不需要花费时间为每一个数据库单独配置查询优化器。我们已经有一组在所有数据库中使用了无数次的初始化参数。"首先，我会这样回答："如果一组单独的参数就可以在所有数据库中运行良好，为什么Oracle还引入了将近二十几个专门针对于查询优化器的初始化参数？它们很擅长自己所做的工作。如果存在这样的一个神奇的配置，它们会默认提供它并隐藏相关的初始化参数。"接下来我会仔细地通过以下两个理由来解释所谓的神奇配置并不存在。

- 每个应用程序都有自己独特的需求和负载情况。
- 每个系统都由不同的硬件和软件组件组成，这些组件都拥有其自己的特征。

如果查询优化器工作良好，就意味着它会为大部分[2]SQL语句生成合理的执行计划。但还要注意，

---

① 肯尼亚谚语的原文是"Peace is costly, but it is worth the expense"，意思是"和平的代价高昂，但是却值得"。详情请参见http://www.quotationspage.com/quote/38863.html。

② 与其他任何你能想象到的活动一样，十全十美在软件开发中也是无法实现的。这条规则，即使你和Oracle都不喜欢，也会适用于查询优化器。因此你应该预料到会有一小部分的SQL语句需要手工介入（详见第11章）。

因为只有在正确配置了查询优化器，并且数据库被设计成能够利用它的全部特性的条件下，上面的话才成立。这一点再怎么强调都不过分。还要注意查询优化器的配置不仅包含初始化参数，而且包括系统统计信息和对象统计信息。

9.2 配置路线图

因为不存在类似神奇配置这样的事情，我们需要一个可靠的、可信的规程来帮助我们。图9-1总结了我所使用的主要步骤。它们的描述如下所示。

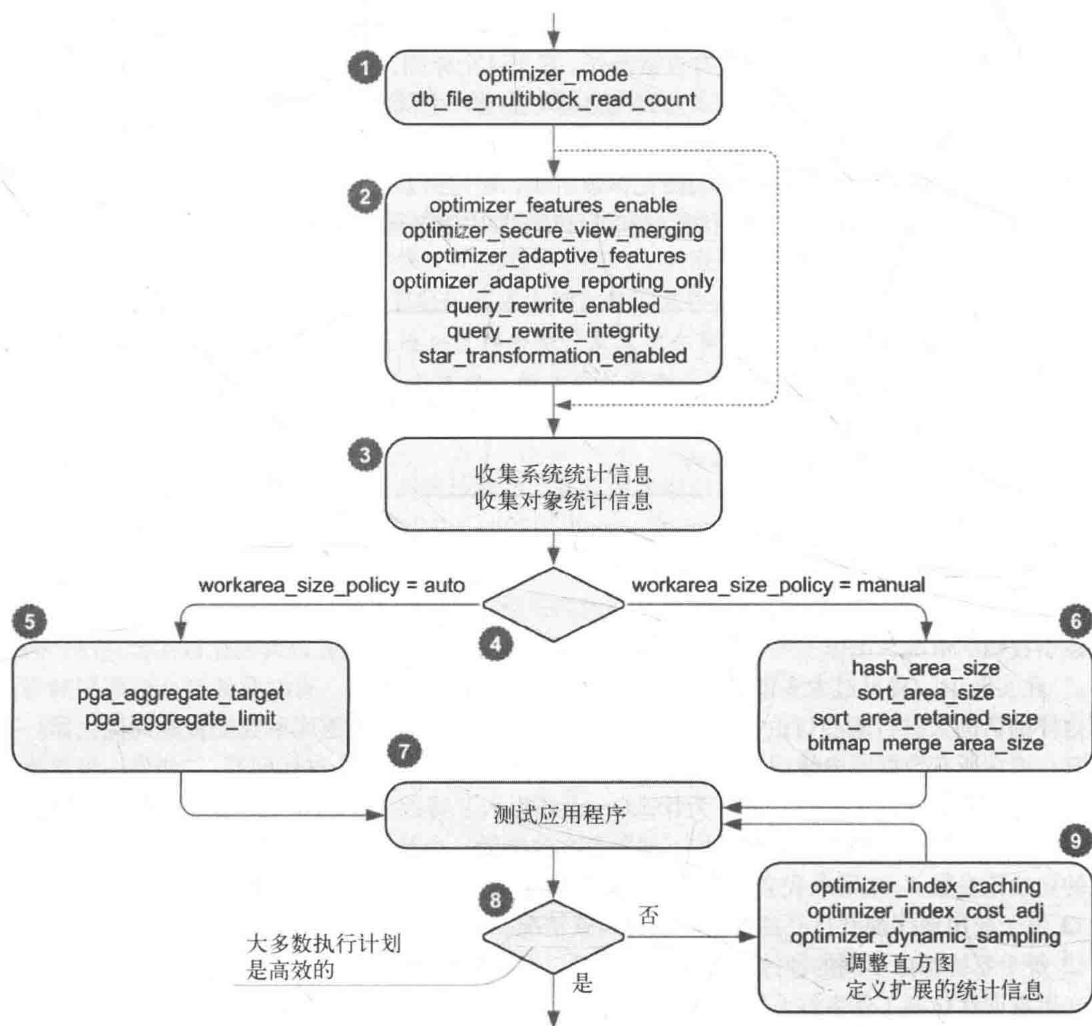


图9-1　配置路线图的主要步骤

(1) 有两个初始化参数需要不断调整：ptimizer_mode和db_file_multiblock_read_count。稍后你会看到，后者并不总是与查询优化器本身直接相关。然而，某些操作的性能可能会极大地依赖于它。

(2) 因为这一步中调整的初始化参数的默认值通常来说工作良好，所以这一步是可选的。不管怎样，这一步的目标是启用或禁用查询优化器的专项特性。

(3) 因为系统统计信息和对象统计信息为查询优化器提供至关重要的信息，所以必须要收集它们。

(4) 通过设置workarea_size_policy初始化参数，可以选择手工还是自动调整在内存中存储数据操作的工作区大小。根据不同的选择方法，其他的初始化参数可以在第5步或第6步中进行设置。

(5) 如果调整工作区大小是自动的，则设置pga_aggregate_target初始化参数。另外，从12.1版本开始，同时还可以修改pga_aggregate_limit初始化参数。

(6) 如果调整工作区大小是手动的，实际大小取决于使用内存的操作的类型。基本上，每种类型的操作都有一个具体的参数。

(7) 当第一部分的配置就位时，就该测试应用程序了。在测试过程中，没有提供需求性能的组件的执行计划被收集起来。通过分析这些执行计划，你应该能够推断出问题所在。注意在这一阶段，重点是识别出普遍行为，而非个例行为。举例来说，你可能注意到查询优化器使用过多或过少的索引或者没有正确识别限制条件。

(8) 如果查询优化器能够为大部分SQL语句生成高效率的执行计划，那表明配置良好。如果不能，继续进行步骤9。

(9) 如果查询优化器趋向于使用过多或者过少的索引或嵌套循环，通常可以通过调整初始化参数optimizer_index_caching和optimizer_index_cost_adj来修复这个问题。如果查询优化器在估算基数方面出现重大错误，可能是因为一些直方图缺失或者需要调整。调整动态采样也可能会有帮助。从11.1版本开始，扩展的统计信息也可能有所帮助。

根据图9-1，步骤1~6中设置的初始化参数不能在事后进行修改。当然了，这也不是一成不变的。如果你无法通过调整与索引相关的初始化参数或步骤9中的直方图来获得理想的结果，可能有必要从头来过。还有必要提一下，因为有几个初始化参数对系统统计信息有影响，在更改完它们之后，可能有必要重新计算系统统计信息。

9.3 设置正确的参数

显然Oracle并不只是随机提供新的初始化参数。相反，引入的每个初始化参数都是为了控制查询优化器的某一特性或者行为。尽管有重复的嫌疑，我还是要提醒你Oracle对于新参数的引进意味着没有一个单独的值能够适用于所用情况。因此，对于每一个初始化参数，必须通过应用程序负载情况和数据库引擎运行的系统共同推断出一个合理的取值。

要为查询优化器执行一个成功的配置，重要的是要理解它是如何运作的，以及每个初始化参数对它的影响。有了这个认识，就不要随意对配置作调整，也不要从最近在网上找到的文章中复制“理想的值”，而应该从以下方面着手。

- 充分了解现状。举例来讲，为什么查询优化器选择了一个非最优的执行计划？
- 决定要实现的目标。换句话说，你要获得什么样的执行计划？
- 找出有哪些初始化参数或者统计信息应该进行调整以达到你设置的目标。当然，在某些情况下仅仅设置初始化参数是不够的。可能有必要修改SQL语句或者数据库设计。

下面的小节将会描述图9-1中配置路线图引用的一些初始化参数如何运作，并给出关于如何为你的

系统找出合理取值的建议。本章没有描述的参数会在本书其他地方讲述这些参数所控制的特性时进行介绍。参数可分成两组：一组参数只影响查询优化器的操作，另一组则与程序全局区（PGA）有关。

9.3.1　查询优化器参数

接下来介绍几个与查询优化器的操作有关的参数。

1. optimizer_mode

optimizer_mode这个参数至关重要，因为通过它可以向查询优化器指明"高效率"这个词的含义。通常来讲，它的意思可能是"更快一些""使用更少的资源"或者其他的意思。因为在使用数据库处理数据时，通常希望处理速度越快越好。因此，高效的含义应该是"用最快的方式执行SQL语句而不浪费不必要的资源"。这对于总是完全执行的SQL语句没有问题（例如，INSERT语句）。而另一方面，对于查询，则会有细微的差异。比如说，应用程序并不是必须要获取查询返回的所有行。换句话说，查询可能是部分执行的。

举一个与Oracle Database无关的例子。当我用谷歌搜索"查询优化器"时，会获得最佳排名的匹配页，首页列出了前十个结果（十个最佳结果）。在同一个页面上，还会显示一条通知消息：在结果集中有986 000个页面，并且搜索它们花费了0.26秒。这是一个优化处理流程以尽可能快速地返回初始数据的例子，因为最前面的几页几乎总是用户唯一真正会去访问的页面。为了访问其中的一页，接下来我单击对应的链接。这时，我通常对于仅获取前几行内容不感兴趣。我希望整个页面都可以访问并且正确排版，也就是说我会开始阅读。在这种情况下，处理流程应该被优化为尽快提供所有数据而非一小部分。每一个应用程序（或程序的一部分）都会归结为以下两种策略：要么是优先快速传递结果集中最靠前的数据，要么是优先快速传递整个结果集（这其实等同于快速传递结果集的最后一行）。

要为optimizer_mode初始化参数选择合适的值，应该首先问自己一个问题：是让查询优化器产生快速返回首行数据的执行计划更重要，还是快速返回末行数据的执行计划更重要。

- 如果快速返回末行数据更重要，应该使用值all_rows 。这是最常用的配置。
- 如果快速返回首行数据更重要，应该使用值first_rows_*n*（*n*的取值为1、10、100或1000）。这个配置应该只在应用程序部分获取的结果集大于该参数指定的行数时才被使用。对于已经存在的应用程序，可以通过比较执行和v$sqlarea视图中的end_of_fetch_count列来检查这一点。注意更早期的首行优化器实现（也就是，通过值first_rows进行配置）不应该再被使用了。事实上，提供这个值仅是为了向后兼容。

默认值是all_rows。还要注意INSERT、DELETE、MERGE和UPDATE语句总是使用all_rows来优化。这样做是很有道理的，因为这些SQL语句必须在将控制权交还给调用者之前处理所有的数据。

警告　首行最优化的关键思想是避免阻塞操作（也就是说，直到运行完毕之前不会产生任何数据的操作）。为此，通常更倾向于嵌套循环连接而非散列连接（直到散列表建立起来之前都是阻塞状态）或合并连接（直到两个输入都完成排序之前都是阻塞状态）。此外，在某些情形中，ORDER BY操作（直到数据被完成排序之前都是阻塞状态）会由索引范围扫描取代。对于大的结果集，首行优化未必能够带来最优的性能表现。所以，最重要的是，只有在调用的应用程序只抓取大结果集的一部分时才使用首行优化。

---

optimizer_mode初始化参数是动态的，并且可以在实例和会话级别修改。在12.1多租户环境下，也可以在PDB级别设置它。此外，通过下面其中一种hint，也可以在语句级别设置它：

- all_rows；
- first_rows(*n*)，*n*是大于0的任何自然数。

2. optimizer_features_enable

在每个数据库版本中，Oracle都会在查询优化器中引入或启用新的特性。如果正在升级到一个新的数据库版本并希望保留查询优化器旧的行为，可以通过将optimizer_features_enable初始化参数设置为升级之前的数据库版本来实现。遗憾的是，并不是所有的新特性都可以通过这个初始化参数来禁用。举例来说，如果你在11.2版本中将其设置为10.2.0.4，就不会获得与10.2.0.4版本完全一样的查询优化器。出于这个原因，我通常建议使用默认值，也就是与数据库可执行文件使用相同的版本号。另外，Oracle Support文档*Use Caution if Changing the OPTIMIZER_FEATURES_ENABLE Parameter After an Upgrade*（1362332.1）也提供了类似建议。

提示 改变optimizer_features_enable初始化参数的默认值只是短期解决方案。迟早应用程序都应该适应（尽量充分利用）新的数据库版本。

---

optimizer_features_enable初始化参数合法的值是类似10.2.0.5、11.1.0.7或 11.2.0.3这样的数据库版本号。因为并不会针对这个参数的补丁级别更新文档（特别是*Oracle Database Reference*手册），所以可以通过以下SQL语句来生成支持的值：

```
SQL> SELECT value
  2  FROM v$parameter_valid_values
  3  WHERE name = 'optimizer_features_enable';

VALUE
----------
8.0.0
8.0.3
8.0.4
...
11.2.0.2
11.2.0.3
11.2.0.3.1
```

optimizer_features_enable初始化参数是动态的，并且可以在实例和会话级别修改。在12.1多租户环境下，也可以在PDB级别设置它。此外，也可以在语句级别通过optimizer_features_enable这个hint来设置一个值。下面的两个例子分别通过这个hint设置默认值和一个具体值（有关hint的详细内容请参见第11章）：

- optimizer_features_enable(default)
- optimizer_features_enable('10.2.0.5')

3. db_file_multiblock_read_count

数据库引擎在多块读取期间（例如，全表扫描或索引快速全扫描）使用的最大磁盘I/O大小是由

db_block_size和db_file_multiblock_read_count初始化参数值的乘积决定的。因此，在多块读取期间读取的最大块数量是由最大磁盘I/O大小除以读取的表空间的块大小来决定的。换句话说，对于默认块大小，db_file_multiblock_read_count初始化参数指定的是读取的最大块数量。这里仅指最大的数量是因为，至少有以下三种常见情况会导致多块读取的数量要小于该初始化参数指定的值。

- 对于段头块和其他只包含像扩展映射这样的段元数据的块，都是通过单块读取的。
- 物理读从来不会横跨多个扩展，但有一个例外，就是针对使用自动段空间管理的表空间执行直接路径读。
- 已经在缓冲区中的数据块，除非是直接路径读，否则不会从磁盘I/O子系统重新读取。

举例说明，图9-2展示了在使用手工段空间管理的表空间中存储的段的结构。与其他任何段一样，它由扩展组成（在本例中有2个），每一个扩展都是由块组成的（在本例中有16个）。第一个扩展的第一个块是段头。某些块（4、9、10、19和21）已经缓存在缓冲区中。为这个段执行缓冲读的数据库引擎进程无法执行任何的物理多块读，即使db_file_multiblock_read_count初始化参数设置为大于或等于32的值也不行。

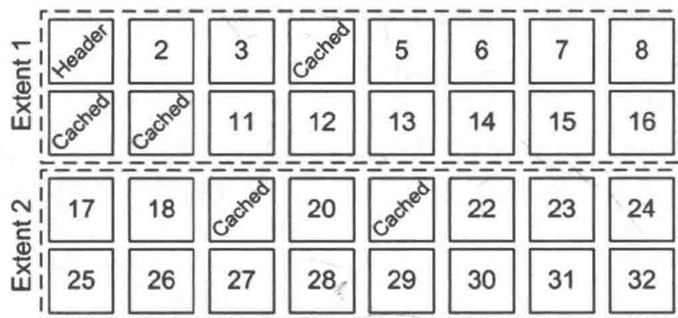


图9-2　数据段的结构

如果将db_file_multiblock_read_count初始化参数设置为8，则会执行下面这些缓冲读。

- 一次段头的单块读（块1）。
- 一次两个块的多块读（块2和块3）。因为块4已经缓存所以无法读取更多的块。
- 一次四个块的多块读（从块5到块8）。因为块9已经缓存所以无法读取更多的块。
- 一次六个块的多块读（从块11到块16）。因为块16是该扩展的最后一个块，所以无法读取更多的块。
- 一次两个块的多块读（块17和块18）。因为块19已经缓存所以无法读取更多的块。
- 一次块20的单块读。因为块21已经缓存所以无法读取更多的块。
- 一次八个块的多块读（从块22到块29）。因为db_file_multiblock_read_count初始化参数被设置为8，所以无法读取更多的块。
- 一次三个块的多块读（从块30到块32）。

概括起来，这个进程执行了两次单块读操作和6次多块读操作。一次多块读读取的平均块数量大概是4个。平均大小小于8的事实解释了为何Oracle会在系统统计信息中引入mbrc值。

db_file_multiblock_read_count初始化参数是动态的，并且可以在实例和会话级别修改。在12.1

多租户环境下，也可以在PDB级别设置它。

这时候，讨论一下查询优化器是如何计算多块读操作（例如，全表扫描或索引快速全扫描）的成本也非常重要。

当有负载系统统计信息可用时，I/O成本并不依赖于db_file_multiblock_read_count初始化参数的值。它是由公式9-1计算而来。注意，之所以用mreadtim除以sreadtim是因为查询优化器根据单块读正常化了成本，就像在第7章中讨论的那样（公式7-2）。

公式9-1 使用有负载统计信息时多块读操作的I/O成本

$$in_cost \approx \frac{blocks}{mbrc} \cdot \frac{mreadtim}{sreadtim}$$

在公式9-1中，若使用无负载统计信息，则变量会替换成以下值。

- 倘若db_file_multiblock_read_count初始化参数明确设置了，则mbrc由db_file_multiblock_read_count初始化参数的值替换；否则，使用8作为值。
- sreadtim由公式7-3计算出来的值计算。
- mreadtim由公式7-4计算出来的值计算。

这意味着只有在使用无负载统计信息时，db_file_multiblock_read_count初始化参数才会对多块读操作的成本产生直接的影响。这还意味着太高的值可能会导致过多的全表扫描或至少造成对多块读操作成本的低估。进一步讲，这是有负载统计信息优于无负载统计信息的另一种情况。

你已经知道了成本公式，现在需要知道如何找出db_file_multiblock_read_count初始化参数应该设置的值。最重要的是要认识到多块读对于性能有重大影响。因此，要小心设置db_file_multiblock_read_count初始化参数的值以达到最佳性能。虽然那些能够引发1 MB磁盘I/O大小的值通常提供近乎最好的性能，但有时高一些或低一些的值会更好。此外，更高的值通常需要更少的CPU来处理磁盘I/O操作。在不同的参数值下执行一个简单的全表扫描，可以给出关于这个初始化参数的影响的有用信息，进而帮助我们找到最佳值。下面的PL/SQL代码段是assess_dbfmbrc.sql脚本的一段摘录，可以用于此用途：

```
BEGIN
  dbms_output.put_line('dbfmbrc blocks seconds cpu');
  FOR i IN 0..10
  LOOP
    l_dbfmbrc := power(2,i);

    EXECUTE IMMEDIATE 'ALTER SESSION SET db_file_multiblock_read_count = '||l_dbfmbrc;
    EXECUTE IMMEDIATE 'ALTER SYSTEM FLUSH BUFFER_CACHE';

    SELECT sum(decode(name, 'physical reads', value)),
           sum(decode(name, 'CPU used by this session', value))
    INTO l_starting_blocks, l_starting_cpu
    FROM v$mystat ms JOIN v$statname USING (statistic#)
    WHERE name IN ('physical reads','CPU used by this session');

    l_starting_time := dbms_utility.get_time();

    SELECT count(*) INTO l_count FROM t;
```

9

```
    l_ending_time := dbms_utility.get_time();

    SELECT sum(decode(name, 'physical reads', value)),
           sum(decode(name, 'CPU used by this session', value))
    INTO l_ending_blocks, l_ending_cpu
    FROM v$mystat ms JOIN v$statname USING (statistic#)
    WHERE name IN ('physical reads','CPU used by this session');

    l_time := round((l_ending_time-l_starting_time)/100,1);
    l_blocks := l_ending_blocks-l_starting_blocks;
    l_cpu := l_ending_cpu-l_starting_cpu;
    dbms_output.put_line(l_dbfmbrc||' '||l_blocks||' '||to_char(l_time)||' '||to_char(l_cpu));
  END LOOP;
END;
```

如你所见，实现起来也没有那么难。无论如何，当心不要在操作系统和磁盘I/O子系统级别缓存测试表，因为那样会导致测试失效。避免这样做的最简单方式是使用比你系统中可用的最大缓冲区还要大的表。对于预计要使用并行处理的系统，也值得去扩展这样的一个测试来执行并行查询（详见第15章）。

图9-3展示了在我的测试系统上所有初始化参数设置为默认值的情况下，针对一个11.2的数据库执行以上的PL/SQL代码块测量得到的特征值。下面是需要注意的特征。

- 吞吐率由db_file_multiblock_read_count取较小值时的200 MB/s增加到使用很大值时的600 MB/s。
- CPU使用率从db_file_multiblock_read_count初始化参数取较小值时的1.5秒下降到取很大值时的0.5秒。

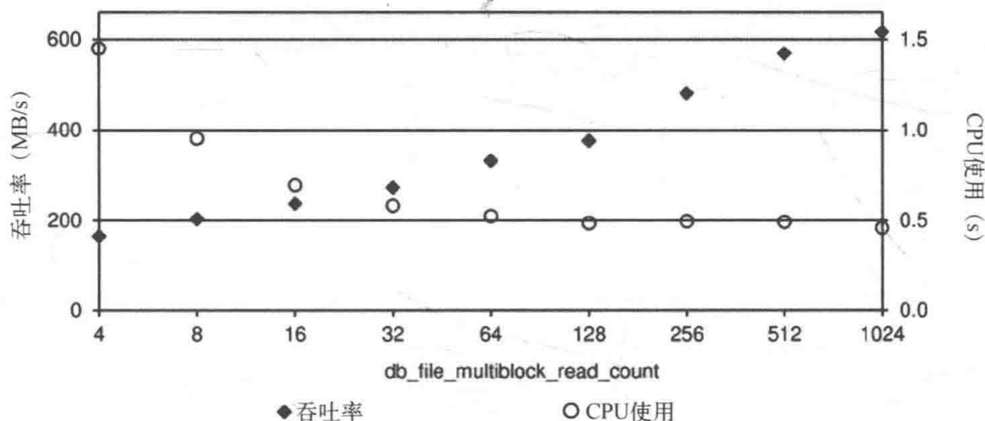


图9-3 磁盘I/O大小对于在四个不同的系统上执行全表扫描的性能的影响

也可以让数据库引擎自动配置db_file_multiblock_read_count初始化参数的值。要使用这个特性，只需不设置它就可以了。如公式9-2所示，接下来数据库引擎就会尝试将其设置为一个能够允许1MB物理读的值。然而，不管怎样，如果缓冲区的大小与数据库支持的会话数量相比非常小，就会应用某种合理性检查以减小这个值。

公式9-2 db_file_multiblock_read_count初始化参数的默认值

$$db_file_multibolck_read_count \approx \text{least}\left(\frac{1048576}{db_block_size}, \frac{db_cache_size}{sessions \cdot db_block_size}\right)$$

正如之前描述的那样，1 MB的物理读并不总是最佳选择，所以建议不要使用这个特性。最好能够具体问题具体分析以找出最合适的值。

要知道如果将无负载统计信息与这个自动配置一起使用，mbrc就不会被公式9-1自动配置的值取代，而是会使用8这个值。

4. optimizer_dynamic_sampling

以往，查询优化器的估算只依靠存储在数据字典中的对象统计信息。有了动态采样，情况就不一样了。事实上，在解析阶段也可能会动态收集某些统计信息。这意味着要收集额外的信息，会针对引用的对象执行一些（采样）查询。遗憾的是，由动态采样收集的统计信息既不会存储在数据字典中，也不会存储在其他什么地方。事实上重用它们的唯一方式就是在共享游标内部重用它们。还要注意由动态采样收集的技术并非一定要使用。实际上，查询优化器会执行一些合理性检查来决定是否应该使用它们。

注意 自12.1版本开始，已使用**动态统计信息**（dynamic statistics）取代了动态采样。在本书中我总是使用旧名称。

---

optimizer_dynamic_sampling初始化参数的值（也叫作级别）指定如何以及何时使用动态采样。表9-1总结了可接受的值和它们的含义。注意其默认值取决于optimizer_features_enable初始化参数。

- 如果将optimizer_features_enable设置为10.0.0或更高，默认值为级别2。
- 如果将optimizer_features_enable设置为9.2.0，默认值为级别1。
- 如果将optimizer_features_enable设置为9.0.1或更低，则禁用动态采样。

表9-1 动态采样的级别及其含义

级别	什么时候使用动态采样	块的数量*
0	禁用动态采样	0
1	动态采样用于没有对象统计信息的表。但是，只有满足以下三个条件时才会发生：表上没有索引，它是连接的一部分（也可以是子查询或不可合并视图），并且该表在高水位线以下拥有的块的数量要比动态采样需要的块数量多	32
2	动态采样用于所有没有对象统计信息的表	64
3	动态采样用于满足级别2标准的所有表，此外，还有那些推测会用于估算谓词选择率的表	32或64
4	动态采样用于满足级别3标准的所有表，此外，还包括在WHERE子句中引用两个或两个以上列的表	32或64
5	同级别4	64
6	同级别4	128
7	同级别4	256

（续）

级　别	什么时候使用动态采样	块的数量*
8	同级别4	1024
9	同级别4	4096
10	同级别4	所有的块
11	查询优化器决定何时以及如何使用动态采样。此级别从11.2.0.4版本开始才可用	自动决定

* 这是当动态采样通过初始化参数或在语句级别的语法中使用hint触发时用于采样的块的数量。对于级别3和级别4，如果对象统计信息可用，则抽取32个块；否则，抽取64个块。当在对象级别的语法中使用hint的时候，以及对于从1到9的级别，块的数量是用下面的公式计算出来的：$32 \cdot 2^{(level-1)}$。

optimizer_dynamic_sampling初始化参数是动态的，并且可以在实例级别以及会话级别进行修改。在12.1多租户环境下，也可以在PDB级别进行设置。此外，也可以通过hintdynamic_sampling在语句级别指定一个值。这个hint支持以下两种语法。

- 语句级别的语法覆盖optimizer_dynamic_sampling初始化参数的值：dynamic_sampling(level)。
- 对象级别的语法只为特定的表触发动态采样：dynamic_sampling(table_alias level)。

警告　在对象级别语法中通过使用hint触发动态采样时，采样总是会发生。换句话说，查询优化器不去检查是否满足在表9-1中提到的规则。但是，根据对象统计信息是否已经可用，采样的统计信息可能会被丢弃掉。所有这些可能都是不必要的间接开支，所以我不推荐使用对象级别的语法。

从11.2版本开始，如果将optimizer_dynamic_sampling初始化参数设置为默认值，则由查询优化器自动决定如何以及何时将动态采样用于并行执行的SQL语句中。这样做是因为并行SQL语句可能会消耗大量的资源，因此，为其获得尽可能好的执行计划非常关键。

查询优化器可以使用动态采样收集两种类型的统计信息。第一种类型包含以下几个方面：

- 一个段高水位线以下的块的数量
- 一张表中行的数量
- 一个列中唯一值的数量
- 一个列中空值的数量

正如你所看到的，第一种类型的统计信息等同于在数据字典中应该已经可用的对应的统计信息。因此，动态采样收集的统计信息只有在对象统计信息缺失或不准确（陈旧）的条件下才有意义。但是要知道，默认情况下，第一种类型的统计信息只会为那些没有对象统计信息的对象进行收集。但是，可以通过指定hintdynamic_sampling_est_cdn(table_alias)强制收集。你可能需要在有统计信息但是统计信息不准确时做这件事。这个hint会在如果不强制就不会收集时强制进行收集。

动态采样收集的第二种类型的统计信息包含以下几项：

- 谓词的选择率
- 连接的基数（仅从12.1版本开始）
- 聚合的基数（仅从12.1版本开始）

因为这些统计信息超出了通过对象统计信息能提供的信息（尽管在某些情形中谓词的选择率可以通过扩展统计信息获得），它们意图增加对象统计信息能够提供的信息。有了它们，查询优化器可能能够执行更好的估算。

下面的例子（11.2.0.3版本中运行的dynamic_sampling_levels.sql脚本生成的摘录）表明在哪种情况下1和4之间的值会引导动态采样发生。用于测试的表通过下面的SQL语句创建。最初，它们没有对象统计信息。注意，t_noidx表和t_idx表唯一的不同是后者有一个主键（因此也就有一个索引）：

```
CREATE TABLE t_noidx (id, n1, n2, pad) AS
SELECT rownum,
       rownum,
       cast(round(dbms_random.value(1,100)) AS VARCHAR2(100)),
       cast(dbms_random.string('p',1000) AS VARCHAR2(1000))
FROM dual
CONNECT BY level <= 1000

CREATE TABLE t_idx (id CONSTRAINT t_idx_pk PRIMARY KEY, n1, n2, pad) AS
SELECT *
FROM t_noidx
```

下面是首次执行的测试查询。它们之间的唯一区别是，第一个引用的是t_noidx表，第二个引用的是t_idx表：

```
SELECT *
FROM t_noidx t1, t_noidx t2
WHERE t1.id = t2.id AND t1.id < 19

SELECT *
FROM t_idx t1, t_idx t2
WHERE t1.id = t2.id AND t1.id < 19
```

如果将级别设置为1，则只会在第一查询中执行动态采样，因为第二个查询引用的表上有索引。下面是为我的测试库上的t_noidx表收集统计信息时执行的递归查询。为了更容易阅读，一些hint被去掉了，并且用字面值替换了绑定变量。注意在执行这个测试查询之前已打开SQL跟踪。接下来我要做的仅仅是观察生成的跟踪文件以找出执行的是哪一个递归SQL语句：

```
SELECT NVL(SUM(C1),0),
       NVL(SUM(C2),0),
       COUNT(DISTINCT C3),
       NVL(SUM(CASE WHEN C3 IS NULL THEN 1 ELSE 0 END),0)
FROM (
  SELECT 1 AS C1,
         CASE WHEN "T1"."ID"<19 THEN 1 ELSE 0 END AS C2,
         "T1"."ID" AS C3
  FROM "CHRIS"."T_NOIDX" SAMPLE BLOCK (20 , 1) SEED (1) "T1"
) SAMPLESUB
```

下面是需要重点关注的内容。

- 查询优化器计算总的行数，在WHERE子句（id < 19）中指定范围内的行数，以及唯一值的数量和id列空值的数量。
- 必须要知晓查询中使用的值。如果使用了绑定变量，查询优化器必须能够窥探绑定变量以便

执行动态采样。

- SAMPLE子句是用来执行采样的。在我的数据库中t_noidx表占用了155个块，所以采样百分比为20%（32/155）。

警告　根据你要处理的数据，可能需要级别6或7来确保动态采样生成有代表性的信息。毕竟，即使是级别7，最多也只抽取256个块。依赖于数据总量和数据分布情况，抽取很少数量的数据块可能不足以正确地代表一张表的整体内容。

如果将级别设置为2，则在两个测试查询中都会执行动态采样，在这个级别，当对象统计信息缺失时总是会使用动态采样。用来为两张表收集统计信息的递归查询和之前展示的语句是相同的。抽取百分比的增加是因为，在这个级别上，它是基于64个块而不是32个。此外， 对于t_idx表，也会执行下面的递归查询。它的目的是通过扫描索引代替之前查询中扫描的表。这么做是因为，在表上执行快速采样可能会漏掉在WHERE子句中谓词指定范围内出现的数据。而如果这些数据存在，在索引上的快速扫描一定会定位到它们：

```
SELECT NVL(SUM(C1),0),
       NVL(SUM(C2),0),
       NVL(SUM(C3),0)
FROM (
  SELECT 1 AS C1,
         1 AS C2,
         1 AS C3
  FROM "CHRIS"."T_IDX" "T1"
  WHERE "T1"."ID"<19
  AND ROWNUM <= 2500
) SAMPLESUB
```

动态采样的下一个级别是3。从这个级别开始，动态采样也用于数据字典中有可用的对象统计信息的情况。在执行进一步的测试之前，通过下面的PL/SQL代码块收集对象统计信息：

```
BEGIN
  dbms_stats.gather_table_stats(ownname    => user,
                                tabname    => 't_noidx',
                                method_opt => 'for all columns size 1');
  dbms_stats.gather_table_stats(ownname    => user,
                                tabname    => 't_idx',
                                method_opt => 'for all columns size 1',
                                cascade    => true);
END;
```

如果将级别设置为3或更高，查询优化器会执行动态采样，然后通过测算表中数据样本的选择率来估算谓词的选择率，而不是使用来自数据字典的统计信息以及可能是硬编码的值。下面的两个查询验证了这一点：

```
SELECT *
FROM t_idx
WHERE id = 19
```

```
SELECT *
FROM t_idx
WHERE round(id) = 19
```

对于第一个查询，查询优化器能根据列统计信息和直方图估算id=19这个谓词的选择率。因此没有必要进行动态采样。相反，对于第二个查询（除非round(id)表达式上有扩展的统计信息存在），查询优化器无法推断出round(id)=19这个谓词的选择率。事实上，列统计信息和直方图只提供关于id列自身的信息，并没有关于舍入值的。下面的查询是用于动态采样的。正如所看到的，它与之前讨论的那个查询有着相同的结构。c2和c3列不同是因为导致动态采样的SQL语句中的WHERE子句不同了。因为一个表达式作用于索引的列（id）上，与t_idx表一样，所以在这个特殊的案例中在索引上没有执行采样：

```
SELECT NVL(SUM(C1),0),
       NVL(SUM(C2),0),
       COUNT(DISTINCT C3)
FROM (
  SELECT 1 AS C1,
         CASE WHEN ROUND("T_IDX"."ID")=19 THEN 1 ELSE 0 END AS C2,
         ROUND("T_IDX"."ID") AS C3
  FROM "CHRIS"."T_IDX" SAMPLE BLOCK (20 , 1) SEED (1) "T_IDX"
) SAMPLESUB
```

如果将级别设置为4或更高，当WHERE子句中引用同一张表中的两个或两个以上列时查询优化器也会执行动态采样。这样做有助于在有相关列的情况下改进估算能力。下面的查询提供了一个这方面的例子。如果你回头查看创建测试表使用的SQL语句，你会注意到id和n1列包含同样的数据：

```
SELECT *
FROM t_idx
WHERE id < 19 AND n1 < 19
```


同样在本例中，查询优化器通过与之前的例子结构相同的查询执行动态采样。同样，主要的区别还是在于引起动态采样的SQL语句的WHERE子句：

```
SELECT NVL(SUM(C1),0),
       NVL(SUM(C2),0)
FROM (
  SELECT 1 AS C1,
         CASE WHEN "T_IDX"."ID"<19 AND "T_IDX"."N1"<19 THEN 1 ELSE 0 END AS C2
  FROM "CHRIS"."T_IDX" SAMPLE BLOCK (20 , 1) SEED (1) "T_IDX"
) SAMPLESUB
```

总结一下，你可以发现级别1和级别2通常没有太大的帮助。事实上，表和索引都应该拥有最新的对象统计信息。一个常见的例外是当临时表包含的临时数据被访问的时候，临时表可以由全局临时表或普通表实现。实际上，对于它们来讲经常没有对象统计信息可供访问。关于临时表的例外情况是，在12.1版本中，你可以利用会话级别统计信息。不管怎样，要知道一个会话可以共享另一个会话解析的游标，即使这一时刻它被使用了，与临时表关联的段包含完全不同的数据集。级别3以及更高的级别对于改进“复杂”谓词的选择率估算非常有用。因此，如果查询优化器因为“复杂的”谓词无法做出正确的估算，请将optimizer_dynamic_sampling初始化参数设置为4或更高的值。否则，就保持默认值吧。此外，在第8章中提到过，从11.1版本开始可以在表达式和列组上收集统计信息。所以在某些情

形下，应该能够避免动态采样。

5. optimizer_index_cost_adj

optimizer_index_cost_adj初始化参数用于改变通过索引扫描的表访问的成本。合法的值为从1到10 000。默认值是100。大于100的值会使索引扫描成本更加高昂并因此倾向于全表扫描。小于100的值会使索引扫描的成本降低。

要理解这个初始化参数对于成本公式的影响，描述查询优化器如何计算与基于索引范围扫描的表访问有关的成本会很有帮助。

索引范围扫描是对多个键值的索引查找。如图9-4所示，执行的操作如下。

(1) 访问索引的根块。

(2) 遍历分支块来定位包含第一个键值的叶子块。

(3) 对于每一个满足搜索条件的键值，执行以下操作：

a. 提取引用数据块的rowid；

b. 访问由rowid引用的数据块。

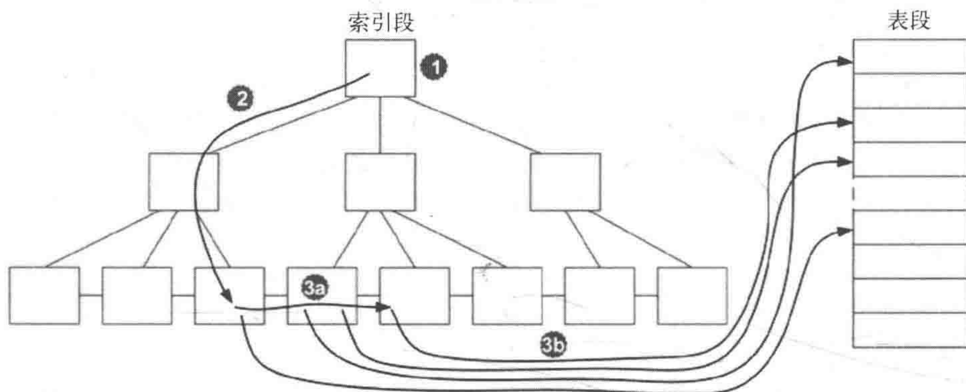


图9-4　在基于索引范围扫描的表访问期间执行的操作

一次索引范围扫描执行的物理读数量等于定位包含第一个键值的叶子块所访问的分支块的数量（也就是blevel统计信息），加上扫描的叶子块数量（leaf_blocks统计信息乘以操作的选择率），再加上通过rowid访问的数据块数量（clustering_factor统计信息乘以操作的选择率）。这样就得到了公式9-3，此外，合并考虑了optimizer_index_cost_adj初始化参数应用的修正。

公式9-3　基于索引范围扫描的表访问的I/O成本

$$in_cost \approx (blevel + (leaf_blocks + cluctering_factor) \cdot selectivity) \cdot \frac{optimizer_index_cost_adj}{100}$$

注意　在公式9-3中，相同的选择率被同时应用于计算索引访问的成本（图9-4中的3a操作）和表访问的成本（3b操作）。在现实中，查询优化器可能会为这两个成本计算使用两个不同的选择率。当只有一部分过滤条件是通过索引访问实施的时候，才有必要这样做。例如，当一个索引由三个列组成而第二个列上没有限制条件时，就会出现这种情况。

概括起来，你可以看到optimizer_index_cost_adj初始化参数对索引访问的I/O成本有着直接的影响。将它设置为一个比默认值小的值时，所有的成本成比例下降。在某些情况下这可能是个问题，因为查询优化器会对其估算的结果进行舍入操作。这就意味着，即使一些索引的对象统计信息是不同的，但是从查询优化器的角度来看它们可能都拥有一样的成本。如果几个成本数值上相等，查询优化器则根据索引的名称决定使用哪一个！它直接按字母顺序选择第一个。在接下来的例子中将演示这个问题。注意当optimizer_index_cost_adj初始化参数和索引名称发生变化时，INDEX RANGE SCAN操作使用的索引是如何变化的。下面是对optimizer_index_cost_adj.sql脚本生成输出的一段摘录：

```
SQL> ALTER SESSION SET OPTIMIZER_INDEX_COST_ADJ = 100;

SQL> SELECT * FROM t WHERE val1 = 11 AND val2 = 11;

-------------------------------------------------
| Id  | Operation                    | Name     |
-------------------------------------------------
|   0 | SELECT STATEMENT             |          |
|*  1 |  TABLE ACCESS BY INDEX ROWID| T        |
|*  2 |   INDEX RANGE SCAN           | T_VAL2_I |
-------------------------------------------------

  1 - filter("VAL1"=11)
  2 - access("VAL2"=11)

SQL> ALTER SESSION SET OPTIMIZER_INDEX_COST_ADJ = 10;

-------------------------------------------------
| Id  | Operation                    | Name     |
-------------------------------------------------
|   0 | SELECT STATEMENT             |          |
|*  1 |  TABLE ACCESS BY INDEX ROWID| T        |
|*  2 |   INDEX RANGE SCAN           | T_VAL1_I|
-------------------------------------------------

  1 - filter("VAL2"=11)
  2 - access("VAL1"=11)

SQL> ALTER INDEX t_val1_i RENAME TO t_val3_i;

SQL> SELECT * FROM t WHERE val1 = 11 AND val2 = 11;

-------------------------------------------------
| Id  | Operation                    | Name     |
-------------------------------------------------
|   0 | SELECT STATEMENT             |          |
|*  1 |  TABLE ACCESS BY INDEX ROWID| T        |
|*  2 |   INDEX RANGE SCAN           | T_VAL2_I |
-------------------------------------------------

  1 - filter("VAL1"=11)
  2 - access("VAL2"=11)
```

要避免这种不稳定性，通常我不推荐将optimizer_index_cost_adj初始化参数设置为较低的值。

同样重要的是，系统统计信息提供与全表范围扫描相关的成本的修正。这就是说，如果系统统计信息就位，默认值通常会表现良好。还要注意系统统计信息没有这个参数所拥有的缺点，因为系统统计信息是增加成本而非降低成本。

optimizer_index_cost_adj初始化参数是动态的，并且可以在实例和会话级别修改。在12.1版本的多租户环境下，也可以在PDB级别设置它。

6. optimizer_index_caching

optimizer_index_caching初始化参数用于指定在in-list迭代操作和嵌套循环连接的执行期间预期在缓冲区中缓存的索引块总量（按百分比算）。应该注意到，这个初始化参数的值仅被查询优化器用来调整它的估算值。换句话说，它并不指定每个索引应该由数据库引擎缓存多少。合法的值范围是从0到100。默认值是0。比0大的值降低in-list迭代操作和嵌套循环连接的内部循环执行的索引扫描的成本。正因如此，optimizer_index_caching参数被用来增加这些操作的使用率。

公式9-4展示了将修正应用于前一小节呈现的索引范围扫描成本公式（公式9-3）后的结果。

公式9-4　基于索引范围扫描的表访问的I/O成本

$$io_cost \approx \left\{ (blevel + leaf_blocks \cdot selectivity) \cdot \left(1 - \frac{optimizer_index_caching}{100}\right) + clustering_factor \cdot selectiuity \right\} \cdot \frac{optimizer_index_cost_adj}{100}$$

这个初始化参数拥有与上一节中描述的optimizer_index_cost_adj初始化参数类似的缺点。虽然如此，它的影响普遍较小主要因为两个原因。首先，它只用于嵌套循环和in-list迭代操作。其次，它对于用于索引范围扫描的成本公式（公式9-4）的群集因子部分没有影响。因为群集因子经常是成本公式中最大的因子，所以这个初始化参数不太可能导致错误的决定。总之，这个初始化参数对于查询优化器的影响比optimizer_index_cost_adj初始化参数要小。也就是说，默认值通常工作良好。

optimizer_index_caching初始化参数是动态的，并且可以在实例和会话级别修改。在12.1版本的多租户环境下，也可以在PDB级别设置它。

7. optimizer_secure_view_merging

optimizer_secure_view_merging初始化参数可以用来控制类似视图合并和谓词迁移之类的查询转换（详见第6章）。可以将它设置为FALSE或TRUE。默认值是TRUE。

- FALSE允许查询优化器无需检查应用查询变换是否会导致安全问题就这样做。
- TRUE允许查询优化器在只有应用查询变换不会导致安全问题时才这样做。

注意　因为名称的原因，你可能会认为optimizer_secure_view_merging初始化参数只与视图合并有关。但是，它控制着所有可能导致安全问题的查询变换。用这个名称的原因很简单：最初实现它时，它只能控制视图合并。

要理解这个初始化参数的影响，我们来看一个例子，该例子演示了为何从安全的角度来看视图合并可能是危险的（完整示例参见optimizer_secure_view_merging.sql脚本）。

假定你有一张很简单的表，该表拥有一个主键和另外两个列：

```
CREATE TABLE t (
  id NUMBER(10) PRIMARY KEY,
  class NUMBER(10),
  pad VARCHAR2(10)
)
```

基于安全的原因，你想要通过下面的视图来提供对这张表的访问。注意通过函数应用的过滤条件来部分地显示这张表的内容。这个函数是如何实现的以及它到底做什么不重要：

```
CREATE OR REPLACE VIEW v AS
SELECT *
FROM t
WHERE f(class) = 1
```

举个例子，一个有权使用这个视图的用户创建了下面的PL/SQL函数。如你所见，它会直接通过对dbms_output包的调用来显示输入参数的值：

```
CREATE OR REPLACE FUNCTION spy (id IN NUMBER, pad IN VARCHAR2) RETURN NUMBER AS
BEGIN
  dbms_output.put_line('id='||id||' pad='||pad);
  RETURN 1;
END;
```

将optimizer_secure_view_merging初始化参数设置为FALSE，可以运行两个测试查询。两个查询都只会返回允许用户查看的那部分值。然而，在第二个查询中，由于视图合并，在查询中添加的函数的执行要早于对函数实施的安全检查。因此，你能够看到你本不能够访问的数据：

```
SQL> SELECT id, pad
  2  FROM v
  3  WHERE id BETWEEN 1 AND 5;

        ID PAD
---------- ----------
         1 DrMLTDXxxq
         4 AszBGEUGEL

SQL> SELECT id, pad
  2  FROM v
  3  WHERE id BETWEEN 1 AND 5
  4  AND spy(id, pad) = 1;

        ID PAD
---------- ----------
         1 DrMLTDXxxq
         4 AszBGEUGEL
id=1 pad=DrMLTDXxxq
id=2 pad=XOZnqYRJwI
id=3 pad=nlGfGBTxNk
id=4 pad=AszBGEUGEL
id=5 pad=qTSRnFjRGb
```

将optimizer_secure_view_merging设置为TRUE，第二个查询返回如下的输出结果。你可以看到，

函数和查询显示了相同的数据：

```
SQL> SELECT id, pad
  2  FROM v
  3  WHERE id BETWEEN 1 AND 5
  4  AND spy(id, pad) = 1;

        ID PAD
---------- ----------
         1 DrMLTDXxxq
         4 AszBGEUGEL
id=1 pad=DrMLTDXxxq
id=4 pad=AszBGEUGEL
```

注意，如果视图的所有者和查询的发起者是同一个用户，optimizer_secure_view_merging初始化参数就会被忽略（因为，阻止一个用户查看他已经可以直接通过查询视图所引用的表来读取的数据，这是毫无意义的）。

一个类似的例子，但是用来展示谓词迁移应用于虚拟私有数据库（VPD）谓词上时的影响，可以在optimizer_secure_view_merging_vpd.sql脚本中找到。

概括起来，通过将optimizer_secure_view_merging初始化参数设置为TRUE，查询优化器检查查询变换是否会导致安全问题。如果会导致这种问题，则查询变换不会被执行，此时性能表现可能不是最优的。基于这个原因，如果你没有将视图也没有将VPD用作安全用途，我建议你将optimizer_secure_view_merging初始化参数设置为FALSE。

optimizer_secure_view_merging初始化参数是动态的，并且可以在实例级别修改。在12.1版本的多租户环境下，也可以在PDB级别进行设置。但是如果用户拥有MERGE VIEW对象权限或者MERGE ANY VIEW系统权限，则不受这个初始化参数施加的限制条件影响。要知道，默认的dba角色提供MERGE ANY VIEW系统权限。

9.3.2 PGA管理

为了执行在内存中存储数据的SQL操作（例如，排序操作和散列联接），会使用工作区。这些工作区在每个服务进程的私有内存（PGA）中进行分配。本节将描述配置这些工作区的初始化参数。

通常，更大的工作区会提供更好的性能。因此，你应该将系统中可用的未分配内存用于工作区的分配中。但是，在修改它的时候要小心。工作区的大小也会对查询优化器的估算产生影响。可以预见的是，改变不仅会体现在性能方面，也会体现在执行计划上。换句话说，如果想避免意想不到的情况，那么所有的修改都应该是经过仔细测试的。

总的来说，本节不会为所描述的初始化参数提供“合理的”值。为某一个应用程序找出合理值的唯一办法，是测试并测量达到合理的性能所需要的PGA的大小。事实上，内存总量只对性能有影响而对一个操作该如何执行没有影响。

1. workarea_size_policy

workarea_size_policy初始化参数指定如何调整工作区大小的工作。可以将它设置为下面两个值中的一个。

- auto：单个工作区的大小调整委托给内存管理器。通过pga_aggregate_target初始化参数，只有整个系统的PGA总量被指定。这是默认值。
- manual：通过hash_area_size、sort_area_size、sort_area_retained_size以及 bitmap_merge_area_size初始化参数，可以完全控制工作区大小的调整。

在大多数情形中，内存管理器运行良好，所以极力推荐将PGA的管理委托给它。只有在很少的情况下手工精心调整可以提供比自动PGA管理更好的结果。

workarea_size_policy初始化参数是动态的，并且可以在实例和会话级别修改。因此可以在系统级别启用自动PGA管理，然后对于特殊要求，在会话级别切换为手工PGA管理。在12.1版本的多租户环境下，也可以在PDB级别进行设置。

2. pga_aggregate_target

如果启用了自动PGA管理，pga_aggregate_target参数指定（按字节）分配给一个数据库实例的PGA总量。支持的值的范围是从10 MB~4 TB。默认值是系统全局区（SGA）大小的20%。对于如何使用这个值很难给出任何具体的建议。但是，在所有的系统上，每个并发的会话至少需要几兆字节的内存。

注意　自11.1版本开始，memory_target和memory_max_target初始化参数可以用于指定一个数据库实例使用的内存总量（也就是SGA大小加上合计的PGA大小）。设置了这两个参数之后，数据库引擎会自动按需要在SGA和PGA之间重新分配内存。在这样的配置中，pga_aggregate_target初始化参数仅用来设置PGA的最小值。

要说明内存管理器是如何工作的，我在11.2.0.3版本中执行了一个需要60 MB左右PGA的查询并逐渐递增并发会话的数量（1~50）。对于每一次迭代，都检查由数据库实例分配的最大PGA总量，并查看由执行查询的会话分配的平均PGA总量。pga_aggregate_target初始化参数被设置为1 GB。这就意味着，如果目标兑现，应该最多有17个会话（1 GB/60 MB）能够获得必要的PGA从而在内存中执行整个语句。图9-5展示了测试的结果。正如你所看到的，数据库实例分配的最大PGA增长了，与配置的一样，达到了1 GB。注意，在第19个并发会话之前，系统PGA与会话数差不多成比例增长。超过17个会话时，系统开始减少提供给每个会话的PGA总量。

一定要理解pga_aggregate_target初始化参数的值并非一个硬性限制，而是更倾向于一个目标值。因此，如果指定的值过低，则数据库引擎可以自由分配比指定的值更多的内存。之所以允许这样做是因为如果无法为操作分配请求的内存则会导致其失败。但是你仍然可以使用pga_aggregate_limit初始化参数（参见下一节）设置一个硬性限制。这个参数从12.1版本开始可用。在这之前的版本中它不可用。

为了展示一个数据库实例过度分配PGA的案例，我通过将pga_aggregate_target初始化参数设置为128 MB重新运行之前的测试。换句话说，我指定的值远远不够运行50个每个都需要60 MB内存的并发会话。图9-6显示了测试的结果。你可以看到，即便是单个会话也无法获取足够的PGA来在内存中执行查询。实际上，那个会话只获得了所需要内存的一半。随着并发会话数量的增加，越来越多的PGA被分配。到第50个会话的时候，使用了大约400 MB的PGA——是配置的目标值的三倍还多。

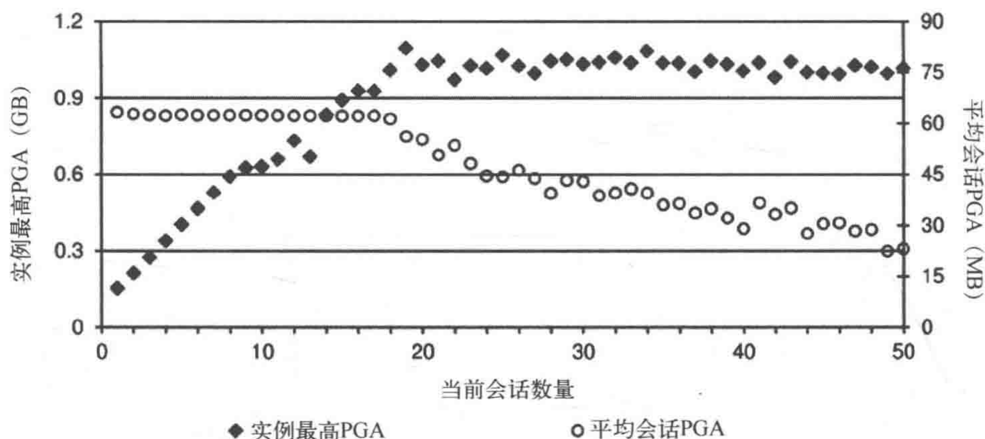


图9-5　内存管理器自动调整提供给会话的PGA总量

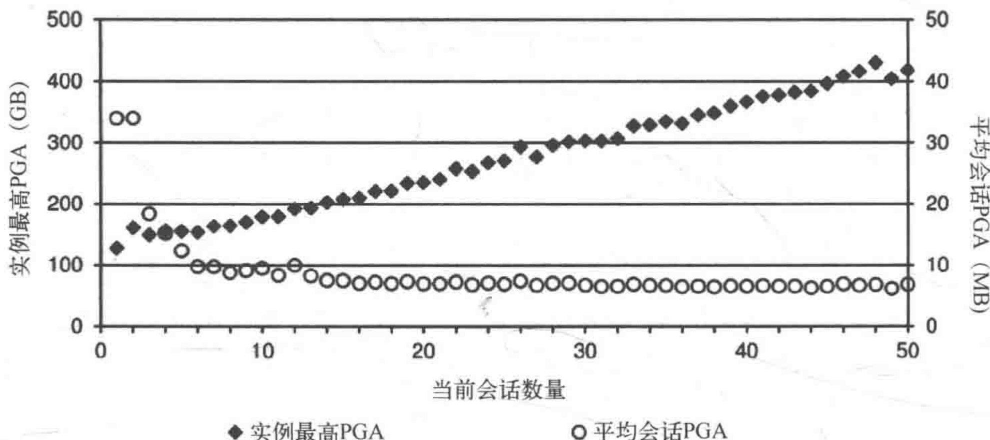


图9-6　如果通过pga_aggregate_target初始化参数设置的目标值（本例中是128 MB）过低，则内存管理器不会遵守它

要了解一个系统是否经历过PGA过度分配的情况，可以使用接下来针对v$pgastat视图的查询。（注意，查询的输出显示的是数据库实例运行完图9-6所示的测试之后的最终状态。）如果像显示的那样，maximum PGA allocated的值远远高于aggregate PGA target parameter的值，就表示pga_aggregate_target初始化参数的值不合适。在这种情况下，重要的是要了解过度分配发生的频率。出于这个目的，over allocation count统计信息表明数据库实例从上一次启动后不得不分配比通过pga_aggregate_target初始化参数指定的值更多的PGA的次数。理想情况下这个值应该是0：

```
SQL> SELECT name, value, unit
  2  FROM v$pgastat
  3  WHERE name IN ('aggregate PGA target parameter',
  4                 'maximum PGA allocated',
  5                 'over allocation count');
```

```
NAME                                VALUE  UNIT
----------------------------- ---------- -----
aggregate PGA target parameter 134217728  bytes
maximum PGA allocated          418658304  bytes
over allocation count                 94
```

你还可以通过v$pgastat视图获得关于当前分配的PGA总量，以及它们中有多少是用于自动或手工工作区的信息。下面的查询说明了这一点。注意尽管拥有total前缀的统计数据提供了当前的使用情况，拥有maximum前缀的统计数据会提供自上一次数据库实例启动以来的最高使用情况：

```
SQL> SELECT name, value, unit
  2  FROM v$pgastat
  3  WHERE name LIKE '% PGA allocated' OR name LIKE '% workareas';

NAME                                       VALUE UNIT
------------------------------------- ---------- -----
total PGA allocated                    999358464 bytes
maximum PGA allocated                 1015480320 bytes
total PGA used for auto workareas      372764672 bytes
maximum PGA used for auto workareas    614833152 bytes
total PGA used for manual workareas            0 bytes
maximum PGA used for manual workareas          0 bytes
```

还要注意，在这个输出当中，分配的PGA只有一部分是用于工作区。很明显，还有其他的东西存储在PGA中。关键点是，每个请求一些内存来执行SQL语句或PL/SQL子程序的进程都能够分配一部分通过pga_aggregate_target初始化参数配置的PGA。即使这些内存在不用于工作区的情况下也可以完成分配。因为内存管理器无法控制这些附加的内存结构（又称为无法调整的内存）的大小，部分PGA也不在内存管理器的控制下。因此，根据系统负载，工作区可用的PGA总量会随时间而变化。在任意给定的时刻可以通过aggregate PGA auto target统计信息查看可用的内存总量。接下来的例子是来自pga_auto_target.sql脚本输出的一段摘录，显示了如何通过PL/SQL调用定义的收集操作来分配500 MB的PGA，进而减少工作区可用的内存总量：

```
SQL> SELECT name, value, unit
  2  FROM v$pgastat
  3  WHERE name LIKE 'aggregate PGA %';

NAME                                VALUE UNIT
----------------------------- ----------- -----
aggregate PGA target parameter 1073741824 bytes
aggregate PGA auto target       910411776 bytes

SQL> execute pga_pkg.allocate(500000)

SQL> SELECT name, value, unit
  2  FROM v$pgastat
  3  WHERE name LIKE 'aggregate PGA %';

NAME                                VALUE UNIT
----------------------------- ----------- -----
aggregate PGA target parameter 1073741824 bytes
aggregate PGA auto target       375754752 bytes
```

```
SQL> execute dbms_session.reset_package;

SQL> SELECT name, value, unit
  2  FROM v$pgastat
  3  WHERE name LIKE 'aggregate PGA %';

NAME                                 VALUE UNIT
------------------------------ ----------- -----
aggregate PGA target parameter 1073741824  bytes
aggregate PGA auto target       910411776  bytes
```

pga_aggregate_target初始化参数是动态的，并且可以在实例级别修改。在12.1版本的多租户环境下，也可以在PDB级别进行设置。

3. pga_aggregate_limit

pga_aggregate_limit初始化参数是12.1版本中最新出现的。它对数据库实例可以使用的PGA总量做出了一个硬性限制。这个参数很有用，因为就像上一节描述的那样，通过pga_aggregate_target初始化参数设置的值只是一个目标值，而非硬性限制。在12.1版本中，如有必要，可以同时指定一个硬性限制。

pga_aggregate_limit初始化参数的默认值被设置为以下值中较大的那一个：

- 2 GB
- pga_aggregate_target初始化参数的值的两倍
- 3 MB乘以processes初始化参数的值

因此，默认情况下会强加一个限制。要避免限制就必须将这个参数设置为0。将这个参数设置为一个比默认值低的值（除了在初始化文件或服务器参数文件中）是不可能的。在尝试设置比默认值低的限制时会引发以下错误：

```
SQL> ALTER SYSTEM SET pga_aggregate_limit = 1G;
ALTER SYSTEM SET pga_aggregate_limit = 1G
*
ERROR at line 1:
ORA-02097: parameter cannot be modified because specified value is invalid
ORA-00093: pga_aggregate_limit must be between 2048M and 100000G
```

当达到限制时，数据库引擎会终止调用甚至是杀掉会话。为了选择要处理的会话，数据库引擎不考虑最大的PGA利用率。相反，数据库引擎会考虑使用最多的不可调整内存总量的会话。当调用被终止时，会引发以下错误：

```
ORA-04036: PGA memory used by the instance exceeds PGA_AGGREGATE_LIMIT
```

当会话被杀掉时，会引发一个典型的ORA-03113错误：

```
ORA-03113: end-of-file on communication channel
Process ID: 5125
Session ID: 17 Serial number: 39
```

此外，会将类似下面这样的对应的信息写入到alert.log中：

```
PGA memory used by the instance exceeds PGA_AGGREGATE_LIMIT of 2048 MB
```

```
Immediate Kill Session#: 17, Serial#: 39
Immediate Kill Session: sess: 0x77eb7478 OS pid: 5125
```

pga_aggregate_limit初始化参数是动态的，并且只能在实例级别更改。在12.1版本的多租户环境下，也可以在PDB级别进行设置。

4. sort_area_size

如果启用了手工PGA管理，sort_area_size初始化参数指定（按字节）用于合并联接、排序以及聚合（包括散列分组）的工作区的大小。注意，这是一个工作区的大小，而一个单独的会话可能会分配多个工作区（详见第14章）。因此，用于整个系统的PGA总量取决于分配的工作区的数量而不是会话的数量。默认值是64 KB。尽管几乎不可能给出关于建议值的一般性建议，默认值确实很小，而通常至少需要使用512 KB/1 MB。值得注意的是，工作区并非总是完全分配的。换句话说，通过sort_area_size初始化参数指定的值只是一个限制。因此，指定一个比实际需要大的值不一定会有问题。

sort_area_size初始化参数是动态的，并且可以在实例和会话级别修改。在12.1版本的多租户环境下，也可以在PDB级别进行设置。

5. sort_area_retained_size

在上一节中，你了解到sort_area_size初始化参数指定用于排序操作的工作区的最大尺寸。尽管严格来说，sort_area_size初始化参数只是指定了当排序操作发生时使用的内存总量。当获得最后一行并将其包含在工作区中存储的已排序结果中后，仍需将内存仅用作将已排序结果返回给父操作的缓冲区。sort_area_retained_size初始化参数指定（按字节）为这个读缓存保留的内存总量。这个初始化参数仅用于启用了手工PGA管理时。尽管默认值是从sort_area_size初始化参数得到的，在v$parameter视图中其显示为0。

要设置这个初始化参数，你必须清楚，如果将它设置为一个比sort_area_size初始化参数低的值，并且结果集无法纳入到保留的内存中，当排序操作完成时数据就会涌入临时段中。即使排序操作本身是完全在内存中执行的，也可能发生这种情况！因此，为了更好的性能而使用默认值是明智的。只有当系统真的在内存上捉襟见肘时，才有理由设置这个参数。

sort_area_retained_size初始化参数是动态的，并且可以在实例和会话级别进行修改。在12.1版本的多租户环境下，也可以在PDB级别进行设置。

6. hash_area_size

如果启用了手工PGA管理，hash_area_size初始化参数指定（按字节）用于散列联接的工作区的大小。要清楚这是一个工作区的大小，而一个单独的会话可能会分配多个工作区。这意味着用于整个系统的PGA总量取决于分配的工作区的数量而非会话的数量。默认值是sort_area_size初始化参数值的两倍。同样，给出具体的建议值非常困难。不管怎样，对于多达4 MB的值，至少应该将其设置为sort_area_size初始化参数值的四到五倍。如果不这样，查询优化器可能会对散列联接的成本评估过高，并因此倾向于为它们使用合并联接。同样，工作区并非总是完全分配的。换句话说，通过hash_area_size初始化参数指定的值只是一个限制值。指定一个比实际需要大的值不一定会有问题。

hash_area_size初始化参数是动态的，并且可以在实例和会话级别进行修改。在12.1版本的多租户环境下，也可以在PDB级别进行设置。

7. bitmap_merge_area_size

如果启用了手工PGA管理，bitmap_merge_area_size初始化参数指定（按字节）用于合并与位图索引关联的位图的工作区大小。默认值是1 MB。再说一次，几乎不可能给出关于建议值的一般性建议。很明显，如果使用了很多位图索引（例如，由于星型转换的原因，参见第14章），则更大的值可能会改进性能。

bitmap_merge_area_size初始化参数是静态参数，并且不能在系统或会话级别进行修改。因此必须要重启数据库实例才能设置它。在12.1版本的多租户环境下，不可以在PDB级别进行设置。

9.4 小结

本章主要讲述如何通过设置初始化参数来实现查询优化器的合理配置。因此，不仅要理解初始化参数是如何工作的，还要理解对象和系统统计信息是如何影响查询优化器的。

即使完成了最佳配置，查询优化器也可能出现无法找出高效执行计划的情况。对一个SQL语句的性能有疑问时，首先需要做的就是审查执行计划。第10章将讨论如何获得执行计划，以及更重要的，如何解释它们，另外也会介绍一些识别低效执行计划的规则。

第 10 章 执行计划

执行计划描述数据库引擎执行SQL语句时实施的操作。每当你不得不去分析一个与SQL语句有关的性能问题时，或者对查询优化器创造条件做出的决定有疑问时，你必须了解执行计划。没有它，你就像一个拿着拐杖的盲人走在撒哈拉大沙漠的中央，四处摸索着试图找出一条出路。当分析或质疑一个SQL语句的性能时，需要做的第一件事就是获取它的执行计划，这再怎么强调都不为过。

无论何时处理一个执行计划，你都需要实施三个基本操作：获取它，解释它，然后评估它的效率。本章的目标是详细描述应该如何执行这三个操作。

10.1 获取执行计划

基本上，Oracle Database提供五种方法来获取与某个SQL语句关联的执行计划。

- 执行EXPLAIN PLAN语句然后查询其输出所写入的表。
- 查询动态性能视图来显示缓存在库缓存中的执行计划。
- 使用实时监控（Real-time Monitoring）来获取关于正在执行或刚刚执行完毕的SQL语句的信息。
- 查询自动工作负载存储库（AWR）或statspack表，显示存储在存储库中的执行计划。
- 激活跟踪功能提供执行计划。

尽管还有其他获取执行计划的方法（例如，在第11章中会讲到，通过与SQL探查和SQL计划基线相关的特性），但是那些方法无法直接用来获取一个与给定SQL语句关联的执行计划。因此，本章不会介绍这些内容。因为所有显示执行计划的工具都是利用刚才所列的五种方法之一，接下来的内容只会讲述基础知识而不会关注某个具体的工具，例如Oracle Enterprise Manager、PL/SQL Developer或Toad等。不讨论这些工具的原因还有，多半情况下，它们无法提供进行一个完全分析所需的全部信息。注意，实时监控已在第4章中提到过。

10.1.1 EXPLAIN PLAN 语句

EXPLAIN PLAN的目标是接受一个SQL语句作为输入，然后提供它的执行计划和相关信息，并在计划表中作为输出显示。换句话说，通过这个语句可以询问查询优化器，什么样的执行计划将用于给定SQL语句的执行。

图10-1展示了EXPLAIN PLAN语句的语法。可用的参数如下。

- statement指定应该为哪一条SQL语句提供执行计划。支持的SQL语句如下：SELECT、INSERT、

UPDATE、MERGE、DELETE、CREATE TABLE、CREATE INDEX以及ALTER INDEX。

- id指定一个名称，用于区分存储在计划表中的多个执行计划。支持30个字符以内的任何字符串。这个参数是可选的，默认值是NULL。
- table指定将关于执行计划的信息插入到的计划表的名称。这个参数是可选的，默认值是plan_table。一旦有需要，也可以使用通常的语法指定一个模式名以及数据库链接名：schema.table@dblink。

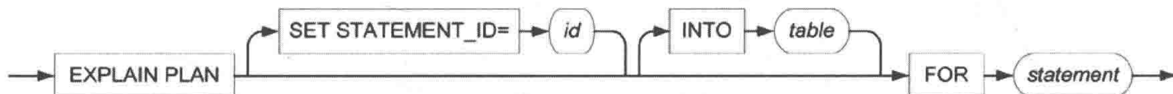


图10-1　EXPLAIN PLAN语句的语法

一定要认识到EXPLAIN PLAN语句是一个DML语句，而非一个DDL语句。这意味着它不会为当前的事务执行一个隐式提交。它只是简单地将数据插入到计划表中。

要执行EXPLAIN PLAN语句，需要将执行SQL语句的权限作为一个参数传递进去。注意当获取视图的执行计划时，同样需要所有底层表和视图的权限。因为这有点违反直觉，看一下下面的例子。注意为何用户能够执行一个引用user_objects视图的查询却不能为相同的查询语句执行EXPLAIN PLAN语句：

```
SQL> SELECT count(*)FROM user_objects;

  COUNT(*)
----------
        29

SQL> EXPLAIN PLAN FOR SELECT count(*)FROM user_objects;
EXPLAIN PLAN FOR SELECT count(*)FROM user_objects
                                           *
ERROR at line 1:
ORA-01039: insufficient privileges on underlying objects of the view
```

就像错误信息中指出的，用户缺少一个或几个被user_objects视图引用的数据字典表的SELECT权限。

1. 计划表

计划表是EXPLAIN PLAN语句输出内容写入的地方。如计划表不存在，则会抛出一个错误。默认的计划表归SYS所有，一个名为plan_table的公共同义词将这张表暴露给所有的用户。一旦需要一张私有的计划表，通过utlxplan.sql脚本手工创建是一个不错的实践，脚本可以在$ORACLE_HOME/rdbms/admin目录下找到。如果计划表是手工创建的，一旦执行了数据库升级，不要忘记将计划表删掉并重新创建。实际上，这往往会发生在新版本添加了新属性的时候。

有趣的是，默认的计划表是一张会将数据存储直到会话结束的全局临时表[1]。通过这种方式，几个并发的用户可以同时使用它而不互相干扰。

要将计划表与EXPLAIN PLAN语句一起使用，至少需要INSERT和SELECT权限。尽管可以不使用DELETE

① 换句话说，它是一张使用on commit preserve rows选项创建的全局临时表。

权限执行基本的操作，但最好还是授予该权限。

在这里我不会完整地描述计划表，原因很简单：你通常不会直接查询它。关于这张表的列的详细描述，请参考*Performance Tuning Guide*（11.2及之前的版本），或*SQL Tuning Guide*（从12.1版本开始）。

2. 查询计划表

很显然，可以直接通过针对计划表发起查询来获取执行计划。但是，使用dbms_xplan包的display函数会简单得多，如接下来的例子所示。可以看到，它的使用非常简单。实际上，为了显示EXPLAIN PLAN语句生成的执行计划，调用这个函数就足够了。注意这个函数返回的值（即一个结果集）是如何通过table函数转换的：

```
SQL> EXPLAIN PLAN FOR SELECT * FROM emp WHERE deptno = 10 ORDER BY ename;

SQL> SELECT * FROM table(dbms_xplan.display);

PLAN_TABLE_OUTPUT
----------------------------------------------------------------------------
Plan hash value: 150391907

----------------------------------------------------------------------------
| Id  | Operation           | Name | Rows | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |    3 |   114 |     3 (34)| 00:00:01 |
|   1 |  SORT ORDER BY      |      |    3 |   114 |     3 (34)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| EMP  |    3 |   114 |     2 (0) | 00:00:01 |
----------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
  2 - filter("DEPTNO"=10)
```

display函数不仅仅限于不带参数的用法。基于这个原因，本章稍后会介绍dbms_xplan包，探索所有的可能性，包括对产生的输出的描述。

3. 绑定变量陷阱

我遇到过的使用EXPLAIN PLAN语句最常见的错误是，指定了一个有别于要分析的语句的SQL语句。当然，那会导致错误的执行计划。因为格式本身对执行计划没有影响，差别通常由替换绑定变量引起。来检查一下接下来的PL/SQL存储过程中查询语句使用的执行计划：

```
CREATE OR REPLACE PROCEDURE p (p_value IN NUMBER)IS
BEGIN
  FOR i IN (SELECT * FROM emp WHERE empno = p_value)
  LOOP
    NULL; -- do something
  END LOOP;
END;
```

常用的技巧是使用字面值替换PL/SQL变量来复制/粘贴查询语句。执行类似这样的SQL语句：

```
EXPLAIN PLAN FOR SELECT * FROM emp WHERE empno = 7788
```

问题是通过使用字面值替换绑定变量，你向查询优化器提交了一条不一样的SQL语句。这种改变可能会对查询优化器做出的决定产生影响。改变是因为SQL概要、存储纲要、SQL计划基线（详见第11章）的存在，或者查询优化器用来估算在WHERE子句中使用的谓词选择率的方法（字面值和绑定变量不是按照相同的方式处理的）。

正确的途径是使用相同的SQL语句。这是可行的，因为绑定变量可以在EXPLAIN PLAN语句中使用。例如，你应该执行类似的SQL语句（注意，p_value PL/SQL变量被:B1绑定变量替换了，因为PL/SQL引擎也会这么做）：

```
EXPLAIN PLAN FOR SELECT * FROM emp WHERE empno = :B1
```

尽管如此，在EXPLAIN PLAN语句中使用绑定变量有两个问题。第一个问题是，默认情况下，绑定变量会被以VARCHAR2类型声明。结果，数据库引擎可能会自动添加一个隐式转换，而那样做会改变执行计划。这点可以通过在dbms_xplan包中的display函数生成的输出的末尾显示的关于谓词的信息来检查。在下面的输出例子中，to_number函数被用于这个目的：

```
SQL> SELECT * FROM table(dbms_xplan.display);

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------------------
Plan hash value: 4024650034
---------------------------------------------------------------------------------------
| Id | Operation                    | Name   | Rows | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT             |        |    1 |    38 |     1   (0)| 00:00:01 |
|  1 |  TABLE ACCESS BY INDEX ROWID| EMP    |    1 |    38 |     1   (0)| 00:00:01 |
|* 2 |   INDEX UNIQUE SCAN          | EMP_PK |    1 |       |     0   (0)| 00:00:01 |
---------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("EMPNO"=TO_NUMBER(:B1))
```

通常，检查是否正确处理了数据类型是很好的做法，比如，通过为原始SQL语句中所有不是VARCHAR2类型的绑定变量使用显式转换。

第二个问题是在EXPLAIN PLAN语句中使用绑定变量时不会使用绑定变量扫视技术。因为这个问题没有解决方案，所以不能保证通过EXPLAIN PLAN语句生成的执行计划就是运行时会选择的执行计划。换句话说，一旦涉及绑定变量，通过EXPLAIN PLAN语句生成的输出是靠不住的。

10.1.2 动态性能视图

以下四个动态性能视图会显示关于出现在库缓存中的游标信息。

- v$sql_plan提供与计划表基本上相同的信息。换句话说，它提供执行计划和由查询优化器提供的其他相关信息。几个用于标识与库缓存中的执行计划关联的游标的列，是这个视图与计划表之间唯一显著的差别。
- v$sql_plan_statistics为v$sql_plan视图中的每一个操作提供执行统计，例如消耗的时间和产生的行数。本质上讲，它提供执行计划的运行时行为。这是非常有用的信息，因为v$sql_plan

视图只显示查询优化器在解析阶段做出的估算和决定。因为执行统计信息的采集可能会引发不可忽略的负载（依赖于执行计划和数据库服务器运行的操作系统，负载也可能是微不足道的），默认情况下不会采集它们。要激活采集，必须将statistics_level初始化参数设置为all，或者必须将gather_plan_statistics这个hint指定在SQL语句中。要知道，因为可能出现的负载，我不推荐在系统级别修改statistics_level初始化参数的默认值。

- v$sql_workarea提供关于执行游标所需的内存工作区的信息。它给出运行时内存以及估算的高效执行操作需要的内存总量信息。
- v$sql_plan_statistics_all将v$sql_plan、v$sql_plan_statistics以及v$sql_workarea视图提供的信息通过一个单独的视图展现出来。通过它，可以避免手工连接多个视图。

库缓存中的游标（因此会在这些动态性能视图中显示）通过两个列来标识：address和child_number。通过address列，可以标识父游标。通过两个列一起，可以标识子游标。更常见的做法是用sql_id列替代address列来标识游标。使用sql_id列的好处是它的值只依赖于SQL语句本身。换句话说，对于一个给定的SQL语句，sql_id永远不变（事实上，sql_id是散列函数应用于SQL语句文本的结果）。而另一方面，address列是一个指向内存中SQL语句的句柄的指针，并会随着时间而改变。

要标识一个游标，基本上来说你会面临两种搜寻方法，要么知道执行SQL语句的会话，要么知道SQL语句的文本。在两种情况下，一旦标识出子游标，就可以显示它的相关信息了。

1. 标识子游标

你必须面对的第一种常见的情况是，试图获取关于与当前连接到实例的会话有关的SQL语句的信息。在这种情况下，可以在v$session视图上执行查找。当前执行的SQL语句是通过sql_id（或sql_address）和sql_child_number列来标识的。最近执行过的SQL语句是通过prev_sql_id（或prev_sql_addr）和prev_child_number列来标识的。为了演示这种方法的使用，我们假设有一个名叫Curtis的用户给你打电话，抱怨说他正在苦等几分钟以前通过一个应用程序提交的一个请求。对于这个问题，直接查询v$session视图很有效，如下例所示。通过查询的输出，你知道当前他正运行着一个SQL语句（否则，status列不会是ACTIVE），并知道与这个会话关联的游标是哪个：

```
SQL> SELECT status, sql_id, sql_child_number
  2  FROM v$session
  3  WHERE username = 'CURTIS';

STATUS  SQL_ID        SQL_CHILD_NUMBER
------- ------------- ----------------
ACTIVE  1scu79x31qavt                1
```

第二种常见的情况是，你知道你想要查找更多信息的那个SQL语句的文本。在这种情况下，可以在v$sql视图上执行查找。与游标有关联的文本可以在sql_text和sql_fulltext列中找到。这两个列的区别是第一个列只通过一个VARCHAR2（1000)的值显示部分的文本，而第二个列通过CLOB类型的值显示全部文本。举例来说，如果你知道所要查找的SQL语句包含一段online discount的文本，就可以使用下面的查询来找出游标的标识符：

```
SQL> SELECT sql_id, child_number, sql_text
  2  FROM v$sql
  3  WHERE sql_fulltext LIKE '%online discount%'
```

```
  4 AND sql_text NOT LIKE '%v$sql%';

SQL_ID        CHILD_NUMBER SQL_TEXT
------------- ------------ --------------------------------------------------
1hqjydsjbvmwq            0 SELECT SUM(AMOUNT_SOLD)FROM SALES S, PROMOTIONS P
                           WHERE S.PROMO_ID = P.PROMO_ID AND PROMO_SUBCATEGORY
                            = 'online discount'
```

2. 查询动态性能视图

要获得执行计划，可直接在v$sql_plan和v$sql_plan_statistics_all视图上执行查询。但是，还有更简单更好的方式来完成这件事：使用dbms_xplan包的display_cursor函数。如下例所示，其用法与之前讨论的调用display函数相似。唯一的区别是将标识要显示的子游标的两个参数传递给这个函数：

```
SQL> SELECT * FROM table(dbms_xplan.display_cursor('1hqjydsjbvmwq', 0));

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------
SQL_ID 1hqjydsjbvmwq, child number 0
-------------------------------------
SELECT SUM(AMOUNT_SOLD)FROM SALES S, PROMOTIONS P WHERE S.PROMO_ID =
P.PROMO_ID AND PROMO_SUBCATEGORY = 'online discount'

Plan hash value: 265338492

-------------------------------------------------------------------------------------
| Id  | Operation               | Name       | Rows | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |            |      |       |   139 (100)|          |
|   1 |  SORT AGGREGATE         |            |    1 |    30 |            |          |
|*  2 |   HASH JOIN             |            | 913K|   26M|   139  (33)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL    | PROMOTIONS |   23 |   483 |     4   (0)| 00:00:01 |
|   4 |    PARTITION RANGE ALL|            | 918K| 8075K|   123  (27)| 00:00:01 |
|   5 |     TABLE ACCESS FULL   | SALES      | 918K| 8075K|   123  (27)| 00:00:01 |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

  2 - access("S"."PROMO_ID"="P"."PROMO_ID")
  3 - filter("PROMO_SUBCATEGORY"='online discount')
```

display_cursor函数并不限于使用两个参数标识一个子游标。出于这个原因，本章稍后会讲到dbms_xplan包，来探索所有的可能性，包括对生成的输出的描述。

10.1.3　自动工作负载存储库和 Statspack

捕获某个快照时，自动工作负载存储库（AWR）和Statspack就能够收集执行计划。为了获取执行计划，会针对上一节中描述的动态性能视图执行查询。一旦可用，则执行计划可能会通过Oracle企业管理器或其他的工具在报告中显示。对于AWR和Statspack，存储库表中存储的执行计划都与v$sql_plan

视图中存储的执行计划有着非常类似的结构。出于这个原因，上一节中描述的技巧在这里也适用。

存储在AWR中的执行计划可以通过dba_hist_sql_plan视图访问（从12.1版本开始，也可以使用cdb_hist_sql_plan视图访问）。要查询它们，dbms_xplan包提供了display_awr函数。与这个包提供的其他函数一样，它的使用简单明了。下面的查询是一个例子（注意用于标识SQL语句而传递给display_awr函数的参数是该语句的sql_id）：

```
SQL> SELECT * FROM table(dbms_xplan.display_awr('1hqjydsjbvmwq'));

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------------
SQL_ID 1hqjydsjbvmwq
--------------------
SELECT SUM(AMOUNT_SOLD)FROM SALES S, PROMOTIONS P WHERE S.PROMO_ID =
P.PROMO_ID AND PROMO_SUBCATEGORY = 'online discount'

Plan hash value: 265338492

-----------------------------------------------------------------------------------
| Id | Operation             | Name       | Rows | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|  0 | SELECT STATEMENT      |            |      |       |   139 (100)|          |
|  1 |  SORT AGGREGATE       |            |    1 |    30 |            |          |
|  2 |   HASH JOIN           |            | 913K|    26M|   139  (33)| 00:00:01 |
|  3 |    TABLE ACCESS FULL  | PROMOTIONS |   23 |   483 |     4   (0)| 00:00:01 |
|  4 |    PARTITION RANGE ALL|            | 918K|  8075K|   123  (27)| 00:00:01 |
|  5 |     TABLE ACCESS FULL | SALES      | 918K|  8075K|   123  (27)| 00:00:01 |
-----------------------------------------------------------------------------------
```

display_awr函数并不只限于使用一个参数来标识SQL语句。出于这个原因，本章稍后会讲到dbms_xplan包，探索所有的可能性，包括对生成的输出的描述。

当使用一个大于或等于6的级别捕获快照时，Statspack将执行计划存储在stats$sql_plan存储库表中。尽管在dbms_xplan包中没有提供具体的函数来查询存储库的表，但是可以利用display函数来显示其中包含的执行计划。可以在display_statspack.sql脚本中找到具体的例子。

此外，对于AWR和Statspack两者，Oracle数据库都提供了实用的脚本，为具体的SQL语句高亮显示一段时间内执行计划的改变和资源消耗的变化。它们的名称分别是awrsqrpt.sql和sprepsql.sql。可以在目录$ORACLE_HOME/rdbms/admin下找到它们。下面是来自awrsqrpt.sql脚本生成的输出的一段摘录。根据输出结果，在分析的时间段内SQL语句的执行计划发生了改变。平均运行时间从第一个的大概8.3秒（16 577/2/1000）到了第二个的3.7秒（14 736/4/1000）左右：

```
SQL ID: 1hqjydsjbvmwq             DB/Inst: DBM11203/DBM11203 Snaps: 576-577
-> 1st Capture and Last Capture Snap IDs
   refer to Snapshot IDs witin the snapshot range
-> SELECT SUM(AMOUNT_SOLD) FROM SALES S, PROMOTIONS P WHERE S.PROMO_ID = ...

    Plan Hash           Total Elapsed                 1st Capture   Last Capture
#   Value                    Time(ms)    Executions       Snap ID        Snap ID
--- ---------------- ---------------- ------------- ------------- --------------
1   2446651477                 16,577             2           577            577
2   265338492                  14,736             4           577            577
--- ----------------------------------------------------------------------------
```

10

```
Plan 1(PHV: 2446651477)
----------------------

Stat Name                                Statement   Per Execution % Snap
---------------------------------------- ---------- -------------- -------
Elapsed Time (ms)                            16,577        8,288.6    50.2
CPU Time (ms)                                16,071        8,035.3    50.9
Executions                                        2            N/A     N/A
Buffer Gets                                 163,606       81,803.0    90.1
Disk Reads                                  161,900       80,950.0    96.0
Parse Calls                                       2            1.0     1.0
Rows                                              2            1.0     N/A
---------------------------------------------------------------------------

-------------------------------------------------------------------------------------
| Id  | Operation               | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |            |       |       |  2798 (100)|          |
|   1 |  SORT AGGREGATE         |            |     1 |    30 |            |          |
|   2 |   NESTED LOOPS          |            |  913K |   26M |  2798  (27)| 00:00:12 |
|   3 |    TABLE ACCESS FULL    | PROMOTIONS |    23 |   483 |     4   (0)| 00:00:01 |
|   4 |    PARTITION RANGE ALL  |            | 39950 |  351K |   121  (27)| 00:00:01 |
|   5 |     TABLE ACCESS FULL   | SALES      | 39950 |  351K |   121  (27)| 00:00:01 |
-------------------------------------------------------------------------------------

Plan 2(PHV: 265338492)
---------------------

Stat Name                                Statement   Per Execution  % Snap
---------------------------------------- ---------- -------------- -------
Elapsed Time (ms)                            14,736        3,684.0    44.6
CPU Time (ms)                                14,565        3,641.2    46.1
Executions                                        4            N/A     N/A
Buffer Gets                                   6,755        1,688.8     3.7
Disk Reads                                    6,485        1,621.3     3.8
Parse Calls                                       1            0.3     0.5
Rows                                              4            1.0     N/A
---------------------------------------------------------------------------

-------------------------------------------------------------------------------------
| Id  | Operation               | Name       | Rows | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |            |      |       |   139 (100)|          |
|   1 |  SORT AGGREGATE         |            |    1 |    30 |            |          |
|   2 |   HASH JOIN             |            | 913K |   26M |   139  (33)| 00:00:01 |
|   3 |    TABLE ACCESS FULL    | PROMOTIONS |   23 |   483 |     4   (0)| 00:00:01 |
|   4 |    PARTITION RANGE ALL  |            | 918K | 8075K |   123  (27)| 00:00:01 |
|   5 |     TABLE ACCESS FULL   | SALES      | 918K | 8075K |   123  (27)| 00:00:01 |
-------------------------------------------------------------------------------------
```

10.1.4 跟踪工具

有几个提供关于执行计划的信息的跟踪工具。遗憾的是，除了SQL跟踪（参见第3章），它们中没

有一个是受官方支持或记录在案的。但它们或许会派上用场，所以我简单介绍其中的两个。

1. 10053事件

如果你正因为查询优化器做出的决定而陷入严重的困境，并且想知道到底是怎么回事，查询优化器跟踪可能会有所帮助。但我提醒你，阅读跟踪文件并不是一件轻松的任务。幸运的是，你不用经常去读那些文件，除非你是真的对查询优化器的内部工作机制感兴趣。

如果想在某时为一个SQL语句生成跟踪文件并希望手动执行这条语句，常见的做法是将它嵌入到下面两条SQL语句中间，从而启用和禁用10053事件。注意跟踪文件只有在执行硬解析时才会生成：

```
ALTER SESSION SET events '10053 trace name context forever'

ALTER SESSION SET events '10053 trace name context off'
```

如果无法手动执行SQL语句，从11.1版本开始你可以通知查询优化器对一个通过具体的sql_id指定的SQL语句在下次发生硬解析时生成一个跟踪文件。要启用或禁用这种行为，可以使用类似下面这样的SQL语句（当然，你需要更改作为参数传递的sql_id）。这种方法的好处是可以让应用程序发出对应的SQL语句。这样会给你一个与在真实执行环境中执行的真正SQL语句一样的跟踪文件。注意这种方法也可以在会话级别使用。直接将ALTER SYSTEM语句用ALTER SESSION语句替换就可以：

```
ALTER SYSTEM SET events 'trace[rdbms.SQL_Optimizer.*][sql:9s5u1k3vshsw4]'

ALTER SYSTEM SET events 'trace[rdbms.SQL_Optimizer.*][sql:9s5u1k3vshsw4] off'
```

如果你想分析与库缓存中存储的一个游标关联的SQL语句，从11.2版本开始，可以利用dbms_sqldiag包中的dump_trace过程。这种方法既不需要执行SQL语句，也不需要知道SQL语句被解析的真实环境。也无需知道与游标关联的绑定变量的值。这个过程会从库缓存中取得它需要的所有信息，并通知查询优化器重新优化SQL语句并转储一个跟踪文件。下面演示一下如何调用它：

```
dbms_sqldiag.dump_trace(
  p_sql_id       => '30g1nn8wdymh3',
  p_child_number => 0,
  p_component    => 'Optimizer',
  p_file_id      => 'test'
);
```

这个过程的输入参数如下所示。

- p_sql_id指定要处理的父游标。
- p_child_number指定子游标号，它与p_sql_id一起，就可以标识要处理的子游标。这个参数是可选的，且默认值为0。
- p_component指定该过程是否转储Optimizer或Compiler跟踪。简单来说，前者模拟设置10053事件，后者在跟踪文件中写入更多的信息。
- p_file_id为tracefile_identifier初始化参数指定一个值。这个参数是可选的，并且默认值为NULL。

与你如何启用跟踪无关，查询优化器会生成包含大量关于它执行的工作信息的跟踪文件。在文件中你会发现由初始化参数、系统统计信息和对象统计信息决定的执行环境，以及为了找出最高效的执行计划而执行的估算信息。描述这个事件生成的跟踪文件的内容超出了本书的范围。如有必要，请参

考下面的资源。

- Wolfgang Breitling的论文：*A Look under the Hood of CBO: The 10053 Event*。
- Oracle Support文档：*CASE STUDY: Analyzing 10053 Trace Files*（338137.1）。
- Jonathan Lewis的著作*Cost-Based Oracle Fundamentals*（Apress，2006）的第14章。

每个服务进程都会将它解析的SQL语句的所有数据写入到自己的跟踪文件中。这不仅意味着跟踪文件可以包含关于多个SQL语句的信息，而且一旦在多个会话中打开跟踪文件的生成，也会出现多个跟踪文件。关于跟踪文件的名称和位置信息，参考3.1.1节的“找到跟踪文件”部分。

2. 10132事件

可以使用10132事件引发一个跟踪文件的生成，其中包含与每次硬解析关联的执行计划。如果想为特定的模块或应用程序保留所有执行计划的历史记录，这会很有帮助。下面的例子展示了在跟踪文件中为每个SQL语句存储的这种类型的信息，主要是SQL语句和它的执行计划（包含关于谓词的信息）。注意这段输出中我裁掉的两个部分，是提供关于执行环境信息的一长串参数和补丁修复：

```
----- Current SQL Statement for this session (sql_id=gbxvdrz7jvt80)-----
SELECT count(n)FROM t WHERE n BETWEEN 6 AND 19
----- Explain Plan Dump -----

-------------------------------------+-----------------------------------+
| Id  | Operation           | Name    | Rows | Bytes | Cost | Time        |
-------------------------------------+-----------------------------------+
| 0   | SELECT STATEMENT    |         |      |       |    2 |             |
| 1   |  SORT AGGREGATE     |         |    1 |    13 |      |             |
| 2   |   TABLE ACCESS FULL | T       |   14 |   182 |    2 |  00:00:01   |
-------------------------------------+-----------------------------------+
Predicate Information:
----------------------
2 - filter(("N">=6 AND "N"<=19))

Content of other_xml column
===========================
  db_version     : 11.2.0.3
  parse_schema   : CHRIS
  dynamic_sampling: 2
  plan_hash      : 2966233522
  plan_hash_2    : 1071362934

  Outline Data:
  /*+
    BEGIN_OUTLINE_DATA
      IGNORE_OPTIM_EMBEDDED_HINTS
      OPTIMIZER_FEATURES_ENABLE('11.2.0.3')
      DB_VERSION('11.2.0.3')
      ALL_ROWS
      OUTLINE_LEAF(@"SEL$1")
      FULL(@"SEL$1" "T"@"SEL$1")
    END_OUTLINE_DATA
  */

Optimizer state dump:
Compilation Environment Dump
```

```
optimizer_mode_hinted                = false
optimizer_features_hinted            = 0.0.0
...
_px_numa_support_enabled             = true
total_processor_group_count          = 1
Bug Fix Control Environment
    fix 3834770 = 1
    fix 3746511 = enabled
...
End of Optimizer State Dump
```

初始化参数和补丁修复信息的列表特别长。出于这个原因，根据你使用的版本，即使最简单的SQL语句也会有大概10~30 KB的数据写入到跟踪文件中。这样一个跟踪文件的生成可能是一个相当大的开销。应该在只有真正需要时才激活10132事件。10132事件可以通过以下方式启用和禁用。

- 为当前会话启用和禁用此事件。

```
ALTER SESSION SET events '10132 trace name context forever'

ALTER SESSION SET events '10132 trace name context off'
```

- 为整个数据库启用和禁用此事件。警告：这样设置不会立即生效，只会对修改后创建的会话起作用。

```
ALTER SYSTEM SET events '10132 trace name context forever'

ALTER SYSTEM SET events '10132 trace name context off'
```

每个服务进程都会将关于它解析的SQL语句的所有数据写入到自己的跟踪文件中。这不仅意味着跟踪文件可以包含关于多个SQL语句的信息，而且一旦在多个会话中打开跟踪文件的生成，也会出现多个跟踪文件。关于跟踪文件的名称和位置信息，参考3.1.1节的“找到跟踪文件”部分。

10.2 dbms_xplan 包

在本章前面你看到了dbms_xplan包可以用来显示存储在多个位置的执行计划：其中包括计划表中的，库缓存中的，AWR中的以及Statspack存储库中的。接下来的章节将会描述此程序包中可用的此类函数。首先，我们来看一下它们生成的输出。

10.2.1 输出

本节的目标是解释由dbms_xplan包中的某些函数返回的输出中包含的信息。为此，我使用了一个输出样例，该样例由dbms_xplan_output.sql脚本生成，包含大部分可用的部分。因为一页书不足以显示所有的信息，所以不是每个部分的所有信息都显示出来了。我只展示了关键信息。如果缺少了什么，我会指出来。还要注意，对于本节提供的大部分信息，案例和进一步的解释会在本章稍后或第四部分中给出。输出的第一部分如下所示：

```
SQL_ID dwnnunj9nuztb, child number 0
-------------------------------------
SELECT t2.* FROM t1, t2 WHERE t1.n = t2.n AND t1.id > :t1_id AND
t2.id BETWEEN :t2_id_min AND :t2_id_max
```

这部分给出关于SQL语句的以下信息。

- sql_id可以锁定父游标。此信息只有当输出是由display_cursor和display_awr函数生成时才可用。
- child number与sql_id一起可以锁定子游标。此信息只有当输出是由display_cursor函数生成的时候才可用。
- SQL语句的文本只有当输出是由display_cursor和display_awr函数生成时才可用。

第二部分展示的是执行计划的散列值，以及在表格中显示的执行计划本身。以下是摘录：

```
Plan hash value: 2539808735

-----------------------------------------------------------------------------------
| Id  | Operation                    | Name  | Rows | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |       |      |       |    15 (100)|          |
|*  1 |  FILTER                      |       |      |       |            |          |
|*  2 |   HASH JOIN                  |       |   14 |  7756 |    15   (7)| 00:00:01 |
|   3 |    TABLE ACCESS BY INDEX ROWID| T2   |   14 |  7392 |     4   (0)| 00:00:01 |
|*  4 |     INDEX RANGE SCAN         | T2_PK |   14 |       |     2   (0)| 00:00:01 |
|*  5 |    TABLE ACCESS FULL         | T1    |  876 | 22776 |    23   (0)| 00:00:01 |
-----------------------------------------------------------------------------------
```

在这个表格中，为每个操作都提供了估算信息和执行统计。表格的列数直接取决于可用信息的总量。举例来说，关于分区的信息，并行处理的信息或执行统计只有在可以访问时才会显示。出于这个原因，由同一个函数使用一模一样的参数而生成的两份输出也可能会不一样。在本例中，你看到的列是默认可用的。表10-1总结了你可能会看见的所有列。

表10-1　包含执行计划的表格的列

列	描　述
基本信息（总是可用）	
Id	执行计划中每一步操作（行）的标识符。如果数字以星号开头，就表示此行的谓词信息稍后可供查看
Operation	要执行的操作，也被称作行源操作（row source operation）
Name	执行操作所针对的对象
查询优化器估算	
Rows and E-Rows	估算的由操作返回的行数
Bytes and E-Bytes	估算的由操作返回的数据总量
TempSpc和E-Temp	估算的操作需要的临时表空间总量（按字节）
Cost (%CPU)	估算的操作成本。以插入方式给出的CPU成本的百分比。注意这个值是在整个执行计划中累积的。换句话说，父操作的成本包含子操作的成本
Time和E-Time	估算的执行操作所需要的时间总和（HH:MM:SS）
分区	
Pstart	要访问的第一个分区的编号。如果这个编号在解析阶段不可知，则值是KEY或INVALID。当第一个分区会在执行阶段确定的时候使用KEY
Pstop	要访问的最后一个分区的编号。如果这个编号在解析阶段不可知，则值是KEY或INVALID。当最后一个分区会在执行阶段确定的时候使用KEY

（续）

列	描 述
并行和分布式处理	
Inst	对于分布式处理，操作使用的DBLINK的名称
TQ	对于并行处理，表队列用于从属进程之间的通信
IN-OUT	并行或分布式操作之间的关系
PQ Distrib	对于并行处理，生产者用于向消费者发送数据的分布规律
运行时统计*	
Starts	特定操作被执行的次数。在某些特别的案例中，这个统计显示的是特定内存结构被访问的次数（就像稍后在10.3.5节中展示的一样）
A-Rows	由操作返回的实际行数
A-Time	执行操作所花费的实际时间总和（HH:MM:SS.FF）
I/O统计*	
Buffers	在执行期间通过逻辑读访问的块数量
Reads	在执行期间通过物理读访问的块数量
Writes	在执行期间通过物理写访问的块数量
内存使用率统计	
OMem	估算的最优化执行需要的内存总量（按字节）
1Mem	估算的一次路径执行需要的内存总量（按字节）
0/1/M	通过optimal/one-pass/multipass模式执行的次数
Used-Mem	操作在最后一次执行期间使用的内存总量（按字节）
Used-Tmp	操作在最后一次执行期间使用的临时空间总量（按千字节）。这个值必须乘以1024才能与其他内存使用量的列保持一致（例如，32 K代表32 MB）
Max-Tmp	操作使用的临时空间最高总量（按千字节）。这个值必须乘以1024才能与其他内存使用量的列保持一致（例如，32 K代表32 MB）

* 只有当执行统计打开时才可用

下面的部分展示了查询块名称和对象别名：

```
Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------

   1 - SEL$1
   3 - SEL$1 / T2@SEL$1
   4 - SEL$1 / T2@SEL$1
   5 - SEL$1 / T1@SEL$1
```

对于执行计划中的每一步操作，可以看到哪一个查询块是与它关联的，或者是在哪个对象上执行的。当SQL语句多次引用同一张表时，这个信息就至关重要了。查询块名称将会在第11章中与hint一同详细讲解。

第四部分展示hint的集合，即概要，这应该足以重现那个特别的执行计划。需要注意的是，概要中并不总是包含所有必要的hint。第11章会解释为什么有些概要不足以重现一个执行计划，并描述怎样才能存储并利用这样的概要，例如存储概要和SQL计划基线：

```
Outline Data
-------------

  /*+
      BEGIN_OUTLINE_DATA
      IGNORE_OPTIM_EMBEDDED_HINTS
      OPTIMIZER_FEATURES_ENABLE('11.2.0.4')
      DB_VERSION('11.2.0.4')
      ALL_ROWS
      OUTLINE_LEAF(@"SEL$1")
      INDEX_RS_ASC(@"SEL$1" "T2"@"SEL$1" ("T2"."ID"))
      FULL(@"SEL$1" "T1"@"SEL$1")
      LEADING(@"SEL$1" "T2"@"SEL$1" "T1"@"SEL$1")
      USE_HASH(@"SEL$1" "T1"@"SEL$1")
      END_OUTLINE_DATA
*/
```

下面这部分只有在查询优化器利用绑定变量扫视时才会显示出来。其中提供了每个绑定变量的数据类型和值：

```
Peeked Binds (identified by position):
--------------------------------------

  1 - :T1_ID (NUMBER): 6
  2 - :T2_ID_MIN (NUMBER): 6
  3 - :T2_ID_MAX (NUMBER): 19
```

下面的部分显示应用了哪个谓词。对于每个操作，都显示了它们是在哪里（行号）以及如何（access、filter或storage）应用的：

```
Predicate Information (identified by operation id):
---------------------------------------------------

  1 - filter(:T2_ID_MIN<=:T2_ID_MAX)
  2 - access("T1"."N"="T2"."N")
  4 - access("T2"."ID">=:T2_ID_MIN AND "T2"."ID"<=:T2_ID_MAX)
  5 - filter("T1"."ID">:T1_ID)
```

尽管访问谓词是用于利用高效的访问结构来定位数据行的（例如，内存中的散列表，如操作2；或一个索引，如操作4），而过滤谓词却只有在数据已经从存储它们的结构中抽取出来以后才会应用。此外，当使用了Exadata存储服务器，存储谓词会指定一个特殊的过滤器用于减轻底层存储子系统的负载。注意在这一部分中既会出现SQL语句自身的谓词，也有可能出现由查询优化器或虚拟私有数据库（VPD）策略产生的谓词。在上面的例子中，你看到以下谓词。

- 第1行的操作检查绑定变量的值是否会导致空结果集。只有满足:T2_ID_MIN<=:T2_ID_MAX谓词时，查询才能返回数据。如果没有满足这个谓词，就不会执行查询操作的其余动作。
- 第2行的散列联接使用"T1"."N"="T2"."N"谓词来联接两个表。换句话说，访问谓词也有可能出现在联接条件上。具体在这个例子中，访问谓词用于指定内存中包含t1表数据的散列表，其散列键为t1.n，是通过由访问t2表返回的列t2.n的值来探测的（散列联接的工作机制将会在第14章中详细讨论）。
- 第4行的索引扫描访问了t_pk索引来查找t1表的id列。在本例中，访问谓词出现在查找执行的键上。

- 在第5行，t1表中的所有行都通过一次全表扫描读取出来。然后，将数据行从块中抽取出来时，"T1"."ID">:T1_ID谓词应用于这些行上。

下面的部分显示了执行所有操作时会将哪些列作为输出返回。以下是摘录：

```
Column Projection Information (identified by operation id):
-----------------------------------------------------------

  1 - "T2"."N"[NUMBER,22], "T2"."ID"[NUMBER,22], "T2"."PAD"[VARCHAR2,1000]
  2 - (#keys=1)"T2"."N"[NUMBER,22], "T2"."ID"[NUMBER,22],
      "T2"."PAD"[VARCHAR2,1000]
  3 - "T2"."ID"[NUMBER,22], "T2"."N"[NUMBER,22], "T2"."PAD"[VARCHAR2,1000]
  4 - "T2".ROWID[ROWID,10], "T2"."ID"[NUMBER,22]
  5 - "T1"."N"[NUMBER,22]
```

在本例中，千万要注意，第3行的表访问返回了id、n和pad列，而第5行的表访问只返回了n这一个列。基于这个原因，估算的由第3行操作返回的每一行（7392/14 = 528字节）数据总量（按字节）要比第5行操作返回的（22 776/876 = 26字节）大得多。第16章会讲述更多关于数据库引擎部分读取一行的能力，以及为何从性能的角度来看，这样做是合理的。

最后，有一部分提供了关于优化阶段、环境或SQL语句本身的提醒和警告：

```
Note
-----
   - dynamic sampling used for this statement (level=2)
```

此处通知你查询优化器使用了动态采样来收集对象统计信息。

10.2.2 display 函数

display函数返回计划表中存储的执行计划。返回值是dbms_xplan_type_table集合的实例。集合的元素是dbms_xplan_type对象类型的实例。这种对象类型的唯一属性，即plan_table_output，其类型是VARCHAR2（300)。此函数有以下输入参数。

- table_name指定计划表的名称。默认值是plan_table 。如果指定了NULL，则使用默认值。
- statement_id指定SQL语句的名称，当执行EXPLAIN PLAN语句的时候，作为一个可选参数给出。默认值是NULL。如果使用了默认值，则显示最近一次插入计划表的执行计划（如果没有指定filter_preds参数）。
- format指定在输出中提供哪些信息。可以使用基本值(basic、typical、serial、all及advanced)，想要精细控制，可以向它们添加额外的修饰符(adaptive、alias、bytes、cost、note、outline、parallel、partition、peeked_binds、predicate、projection、remote、report及rows)。如果有应该添加的信息，可以选择使用+这个字符作为前缀的修饰符（例如，basic +predicate）。如果有应该移除的信息，可以选择使用-这个字符作为前缀的修饰符（例如，typical -bytes）。可以同时指定多个修饰符（例如，typical +alias -bytes -cost）。表10-2和表10-3分别完整描述了基本值和修饰符。默认值是typical。
- filter_preds指定查询计划表时应用的限制条件。此限制条件为基于计划表中的一个列的常规SQL谓词（例如，statement_id = 'test3'）。默认值是NULL。如果使用了默认值，则会显示最近一次插入到计划表的执行计划。

表10-2 format参数接受的基本值

值	描述
basic	只显示最少的信息量，基本上只有操作和执行所针对的对象
typical	显示最常见的信息，基本上包含所有的信息，除了别名、概要、扫视的绑定变量、子计划、列投影以及报告模式信息
serial	与typical类似，除了关于并行处理的信息没有显示
all	显示除了概要、扫视的绑定变量、子计划以及报告模式信息以外的所有可用信息
advanced	显示除了子计划和报告模式信息之外的所有可用信息

表10-3 format参数接受的修饰符

值	描述
adaptive	控制子计划的显示。这部分在之前的例子中没有展示过，可参考本章稍后的10.3.9节。这个修饰符仅从12.1版本开始才可用
alias	控制包含查询块名称和对象别名部分的显示
bytes	控制执行计划表中Bytes列的显示
cost	控制执行计划表中Cost列的显示
note	控制包含注释部分的显示
outline	控制包含概要部分的显示
parallel	控制并行处理信息的显示，特别是指执行计划表中的TQ、IN-OUT和PQ Distrib列。这些列在之前的例子中没有显示过
partition	控制分区信息的显示，明确地说是执行计划表中的Pstart和Pstop列。这些列在之前的例子中没有显示过
peeked_binds	控制包含扫视的绑定变量部分的显示
predicate	控制包含过滤谓词、访问谓词和存储谓词的部分的显示
projection	控制包含列投影信息的部分的显示
remote	控制远程执行的SQL语句的显示。这部分在之前的例子中没有显示
report	控制报告模式的激活。启用时，关于自适应和重新优化执行计划的额外信息就会显示出来。这部分在之前的例子中没有显示，可参见10.3.9节。这个修饰符仅从12.1版本开始才可用
rows	控制执行计划表中Rows列的显示

要使用display函数，调用者只需要在该包上有EXECUTE权限并在计划表上拥有SELECT权限。

下面的查询显示了相同的执行计划，展示了在基本值basic、typical以及advanced之间的主要区别。以下是对display.sql脚本生成输出的一段摘录：

```
SQL> SELECT * FROM table(dbms_xplan.display(NULL, NULL, 'basic'));

PLAN_TABLE_OUTPUT
----------------------------------

Plan hash value: 2966233522

----------------------------------
| Id | Operation          | Name |
----------------------------------
|  0 | SELECT STATEMENT   |      |
```

```
|   1 |  SORT AGGREGATE    |      |
|   2 |   TABLE ACCESS FULL| T    |
-----------------------------------

SQL> SELECT * FROM table(dbms_xplan.display(NULL, NULL, 'typical'));

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------

Plan hash value: 2966233522

---------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |     4 |     2   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE    |      |     1 |     4 |            |          |
|*  2 |   TABLE ACCESS FULL| T    |    15 |    60 |     2   (0)| 00:00:01 |
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("N"<=19 AND "N">=6)

SQL> SELECT * FROM table(dbms_xplan.display(NULL, NULL, 'advanced'));

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------

Plan hash value: 2966233522

---------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |     4 |     2   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE    |      |     1 |     4 |            |          |
|*  2 |   TABLE ACCESS FULL| T    |    15 |    60 |     2   (0)| 00:00:01 |
---------------------------------------------------------------------------

Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------

  1 - SEL$1
  2 - SEL$1 / T@SEL$1

Outline Data
-------------

  /*+
    BEGIN_OUTLINE_DATA
    FULL(@"SEL$1" "T"@"SEL$1")
    OUTLINE_LEAF(@"SEL$1")
    ALL_ROWS
    DB_VERSION('11.2.0.3')
```

10

```
      OPTIMIZER_FEATURES_ENABLE('11.2.0.3')
      IGNORE_OPTIM_EMBEDDED_HINTS
      END_OUTLINE_DATA
    */

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("N"<=19 AND "N">=6)

Column Projection Information (identified by operation id):
-----------------------------------------------------------

   1 - (#keys=0) COUNT(*)[22]
```

下面的查询展示了如何使用修饰符从基本值basic和typical生成的默认输出中添加或移除信息。因为它们基于与之前例子相同的查询，你可以对比输出结果来查看有何不同之处。以下是一段来自display.sql脚本输出的摘录：

```
SQL> SELECT * FROM table(dbms_xplan.display(NULL, NULL, 'basic +predicate'));

PLAN_TABLE_OUTPUT
-----------------------------------

Plan hash value: 2966233522

-----------------------------------
| Id  | Operation           | Name |
-----------------------------------
|   0 | SELECT STATEMENT    |      |
|   1 |  SORT AGGREGATE     |      |
|*  2 |   TABLE ACCESS FULL| T    |
-----------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("N"<=19 AND "N">=6)

SQL> SELECT * FROM table(dbms_xplan.display(NULL, NULL, 'typical -bytes -note'));

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------

Plan hash value: 2966233522

-----------------------------------------------------------------
| Id  | Operation           | Name | Rows | Cost (%CPU)| Time     |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |    1 |     2   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE     |      |    1 |            |          |
|*  2 |   TABLE ACCESS FULL| T    |   15 |     2   (0)| 00:00:01 |
-----------------------------------------------------------------
```

```
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("N"<=19 AND "N">=6)
```

将current_schema会话参数设置为一个拥有默认名称的计划表的模式时，如果你使用EXPLAIN PLAN语句和display函数，则必须在EXPLAIN PLAN语句的INTO子句中和display函数的table_name参数中加入该模式名称。如果不这么做就会导致display函数引发一个错误。下面的例子演示了该行为：

```
SQL> ALTER SESSION SET current_schema = franco;

SQL> EXPLAIN PLAN FOR SELECT * FROM t;

SQL> SELECT * FROM table(dbms_xplan.display);

PLAN_TABLE_OUTPUT
-----------------------------------------------------

Error: cannot fetch last explain plan from PLAN_TABLE

SQL> EXPLAIN PLAN INTO franco.plan_table FOR SELECT * FROM t;

SQL> SELECT * FROM table(dbms_xplan.display(table_name=>'franco.plan_table'));

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------

Plan hash value: 3956160932

--------------------------------------------------------------------------
| Id  | Operation         | Name | Rows | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |   14 |  1218 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| T    |   14 |  1218 |     3   (0)| 00:00:01 |
--------------------------------------------------------------------------
```

通过display函数也可以查询一张拥有基于v$sql_plan_statistics_all视图结构的计划表。在想要保存那些被有意设计为只在库缓存中短暂存在的信息时，这个特性就派上用场了。因为这样的计划表中包含额外的信息，当通过display函数查询它时，format参数支持接下来的部分描述的额外修饰符，详见表10-4。下面的例子展示了如何利用这个特性保存关于执行最后一条SQL语句时的信息：

```
SQL> SELECT /*+ gather_plan_statistics */ count(*) FROM t;

  COUNT(*)
----------
      1000

SQL> CREATE TABLE my_plan_table
  2  AS
  3  SELECT cast(1 AS VARCHAR2(30)) AS plan_id, p.*
  4  FROM v$sql_plan_statistics_all p
  5  WHERE (sql_id, child_number)= (SELECT prev_sql_id, prev_child_number
  6                                 FROM v$session
```

```
  7                                 WHERE sid = sys_context('userenv','sid'));

SQL> SELECT * FROM table(dbms_xplan.display('my_plan_table', NULL, 'iostats'));

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------

Plan hash value: 2966233522

-------------------------------------------------------------------------------------
| Id  | Operation          | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers | Reads  |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      2 |        |      2 |00:00:00.01 |      10 |      4 |
|   1 |  SORT AGGREGATE    |      |      2 |      1 |      2 |00:00:00.01 |      10 |      4 |
|   2 |   TABLE ACCESS FULL| T    |      2 |   1000 |   2000 |00:00:00.01 |      10 |      4 |
-------------------------------------------------------------------------------------
```

表10-4　format参数接受的修饰符

值	描　述
allstats	这是iostats memstats的简写
iostats	控制运行时统计信息的显示（列Starts、A-Rows、A-Time）、估算的行数（列E-Rows）以及磁盘I/O统计信息（列Buffers、Reads、Writes）
last	默认情况下，allstats、iostats、memstats和rowstats修饰符都会显示所有执行的累积统计信息。如果将这个值加入它们，则仅会显示最后一次执行的统计信息。为并行处理的SQL语句指定的这个修饰符可能并不会像你期望的那样工作。关于这方面的更多信息请参考15.3节
memstats	控制内存使用的统计信息的显示（列OMem、1Mem、O/1/M、Used-Mem、Used-Tmp、和Max-Tmp）
rowstats	控制行计数统计信息的显示（列Starts、E-Rows和A-Rows）。这个修饰符仅从11.2.0.4版本开始才可用
runstats_last	与iostats last一样。这个参数已经不推荐使用，并只是为了向后兼容而提供
runstats_tot	和iostats一样。这个参数已经不推荐使用，并只是为了向后兼容而提供

10.2.3 display_cursor 函数

display_cursor函数返回库缓存中存储的执行计划。注意，在Real Application Clusters环境中，是无法获得远端实例中存储的执行计划的。与display函数一样，其返回值是dbms_xplan_type_table集合的实例。该函数的输入参数如下所示。

- sql_id指定返回的执行计划的父游标。默认值是NULL。如果使用了默认值，就会返回当前会话执行的最后一个SQL语句的执行计划。
- cursor_child_no指定子游标号，它与sql_id一起，确定返回哪个子游标的执行计划。默认值是0。如果指定了NUll，则会返回sql_id参数指定的父游标下的所有子游标。
- format指定显示哪些信息。支持的值与display函数的format参数支持的值相同。此外，如果可以访问执行统计信息（换句话说，如果将statistics_level初始化参数设置为all或在SQL语句中指定了gather_plan_statistics这个hint），那么表10-4中描述的修饰符也同样受支持。默认值为typical。

警告 正如第2章中指出的那样，有时候v$sql视图中的sql_id和child_number列并不足以确定一个子游标。在这种情况下，因为bug 14585499，并且在11.2.0.3及之前的版本中，display_cursor函数会返回错误的数据。要识别出这个问题，请在display_cursor的输出中查找以下错误信息：

```
An uncaught error happened in prepare_sql_statement : ORA-01422: exact fetch returns more than
requested number of rows
```

可以通过display_cursor_ora-01422.sql脚本来重现这个bug。

要使用display_cursor函数，调用者需要在以下动态性能视图上拥有SELECT权限：v$session、v$sql、v$sql_plan以及v$sql_plan_statistics_all。其中select_catalog_role角色和select any dictionary系统权限提供了这些权限。

注意 表10-4中列举的修饰符对于与查询优化器估算有关的以下列有移除的副作用：Bytes、TempSpc、Cost（%CPU）、Time。如果你希望其中一列出现在输出中，必须通过基本值或修饰符明确指定。

下面的例子展示一个查询使用hint gather_plan_statistics来启用执行统计信息的生成。然后会通知display_cursor函数显示最后一次执行的磁盘I/O统计信息。因为没有物理读或写发生，所以仅会显示逻辑读（Buffers）。以下是一段来自display_cursor.sql脚本输出的摘录：

```
SQL> SELECT /*+ gather_plan_statistics */ count(pad)
  2  FROM (SELECT rownum AS rn, pad FROM t ORDER BY n)
  3  WHERE rn = 1;

COUNT(PAD)
----------
         1

SQL> SELECT * FROM table(dbms_xplan.display_cursor('d5v0dt28fp5fh', 0, 'iostats last'));

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------------------

SQL_ID d5v0dt28fp5fh, child number 0
-------------------------------------
SELECT /*+ gather_plan_statistics */ count(pad) FROM (SELECT rownum AS
rn, pad FROM t ORDER BY n)WHERE rn = 1

Plan hash value: 2545006537

---------------------------------------------------------------------------------------
| Id  | Operation             | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
---------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |      |      1 |        |      1 |00:00:00.02 |     147 |
|   1 |  SORT AGGREGATE       |      |      1 |      1 |      1 |00:00:00.02 |     147 |
|*  2 |   VIEW                |      |      1 |   1000 |      1 |00:00:00.02 |     147 |
|   3 |    SORT ORDER BY      |      |      1 |   1000 |   1000 |00:00:00.02 |     147 |
|   4 |     COUNT             |      |      1 |        |   1000 |00:00:00.01 |     145 |
|   5 |      TABLE ACCESS FULL| T    |      1 |   1000 |   1000 |00:00:00.01 |     145 |
---------------------------------------------------------------------------------------
```

```
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("RN"=1)
```

10.2.4　display_awr 函数

display_awr函数返回AWR中存储的执行计划。与display函数一样，其返回值是dbms_xplan_type_table集合的实例。该函数的输入参数如下所示。

- sql_id指定返回哪条SQL语句的执行计划。这个参数没有默认值。
- plan_hash_value指定要返回的执行计划的散列值。默认值是NULL。如果使用了默认值，则会返回所有与sql_id参数确定的SQL语句有关的执行计划。
- db_id指定应该返回哪个数据库上执行的执行计划。默认值是NULL。如果使用了默认值，则使用当前数据库。
- format指定显示哪些信息。尽管在display函数的format参数中使用的值也同样受支持，但并不是所有的信息都能够显示出来。举例来说，因为AWR不存储有关谓词的信息，所以输出中缺少这部分。默认值是typical。

要使用display_awr函数，调用者至少应在以下数据字典视图上拥有SELECT权限：dba_hist_sql_plan和dba_hist_sqltext。如果使用了db_id参数，还需要v$database视图上的SELECT权限。其中select_catalog_role角色提供了这些权限。

下面的查询展示了对于一个给定的SQL语句存在多个执行计划时plan_hash_value参数的用途。注意第一个查询返回了两个执行计划，而第二个查询只返回了一个。以下是一段来自display_awr.sql脚本输出的摘录：

```
SQL> SELECT * FROM table(dbms_xplan.display_awr('48vuyqjwpf9wg', NULL, NULL, 'basic'));

PLAN_TABLE_OUTPUT
-----------------------------------

SQL_ID 48vuyqjwpf9wg
--------------------
SELECT COUNT(N) FROM T

Plan hash value: 2966233522

-----------------------------------
| Id  | Operation          | Name |
-----------------------------------
|   0 | SELECT STATEMENT   |      |
|   1 |  SORT AGGREGATE    |      |
|   2 |   TABLE ACCESS FULL| T    |
-----------------------------------

SQL_ID 48vuyqjwpf9wg
--------------------
SELECT COUNT(N) FROM T
```

```
Plan hash value: 3776247601

-------------------------------------
| Id  | Operation              | Name |
-------------------------------------
|   0 | SELECT STATEMENT       |      |
|   1 |  SORT AGGREGATE        |      |
|   2 |   INDEX FAST FULL SCAN| I    |
-------------------------------------

SQL> SELECT * FROM table(dbms_xplan.display_awr('48vuyqjwpf9wg', 2966233522, NULL, 'basic'));

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------

SQL_ID 48vuyqjwpf9wg
--------------------
SELECT COUNT(N) FROM T

Plan hash value: 2966233522

-----------------------------------
| Id  | Operation          | Name |
-----------------------------------
|   0 | SELECT STATEMENT   |      |
|   1 |  SORT AGGREGATE    |      |
|   2 |   TABLE ACCESS FULL| T    |
-----------------------------------
```

有几种情况会导致一个给定的SQL语句存在多个执行计划，比如添加了一个索引或者只是因为数据（并且进而其对象统计信息）发生了变化。基本上，每次查询优化器执化的环境发生变化，都可能会生成不同的执行计划。当你对一条SQL语句的性能产生疑问，而且认为该SQL在之前一段时间内的运行都没有问题时，这样的输出就有用处了。思路是，检查经过一段时间后，是否使用了多个执行计划执行过该SQL语句。如果是这样，基于可用的信息推断导致这种变化的原因可能是什么。

10

10.3 解释执行计划

我总是很惊讶关于如何阅读执行计划的文档是如此之少，甚至好像有很多人无法正确阅读它们。在这里我尝试通过描述我阅读执行计划时使用的方法来解决这个问题。注意这里不会提供关于不同操作的细节，而是会提供所需要的基础知识，以便于理解如何阅读执行计划。我会在第四部分给出关于大部分常见操作的详细信息。

警告 并行处理会使执行计划的解释更加困难。原因很简单：多个操作并发执行。为了保持叙述尽可能简单，本节并不涵盖并行处理的内容。关于并行处理执行计划的信息会在第15章中提供。

10.3.1 父-子关系

执行计划是一棵树，用来描述SQL引擎执行操作的顺序以及各个操作之间的关系。树中的每个节点是一个行源操作（实际上是作为用C语言编写的一个函数执行的），例如，表扫描、联接或排序。在各操作（节点）之间，存在着父子关系。理解这些关系对于正确阅读执行计划非常关键。当执行计划以文本格式显示时，控制父-子关系的规则如下所示。

- 一个父操作拥有一个或多个子操作。
- 一个子操作只有一个父操作。
- 唯一没有父操作的操作是树的根操作（顶层操作）。
- 子操作跟随着它们的父操作，在右侧缩进排列。依赖于显示执行计划使用的方法，缩进可以是一个空格字符、两个空格或其他什么。这真的不重要。关键是同一个父操作下的所有子操作都拥有相同的缩进。
- 父操作在子操作之前出现（父操作的ID比子操作的ID要小）。如果一个子操作前面有多个与父操作一样缩进的操作，则距离最近的操作为父操作。

接下来是一个由relationship.sql脚本生成的样例执行计划。注意，尽管只有Operation列需要贯穿整个执行计划，Id列出现在这里是为了帮助你更容易地标识操作。用来生成它的SQL语句被有意忽略掉，因为它并不服务于本节的内容：

```
---------------------------------------------
| Id  | Operation                    | Name |
---------------------------------------------
|   0 | UPDATE STATEMENT             |      |
|   1 |  UPDATE                      | T    |
|   2 |   NESTED LOOPS               |      |
|   3 |    TABLE ACCESS FULL         | T    |
|   4 |    INDEX UNIQUE SCAN         | T_PK |
|   5 |   SORT AGGREGATE             |      |
|   6 |    TABLE ACCESS BY INDEX ROWID| T    |
|   7 |     INDEX FULL SCAN          | I    |
|   8 |   TABLE ACCESS BY INDEX ROWID | T    |
|   9 |    INDEX UNIQUE SCAN         | T_PK |
---------------------------------------------
```

图10-2提供了执行计划的图示。使用之前描述的规则，你可以推断出以下内容。

- 操作0是这棵树的根。它告知你执行计划关联的SQL语句的类型。操作0有一个子操作：操作1。
- 操作1有三个子操作：2、5，还有8。
- 操作2有两个子操作：3和4。
- 操作3和4没有子操作。
- 操作5有一个子操作：6。
- 操作6有一个子操作：7。
- 操作7没有子操作。
- 操作8有一个子操作：9。
- 操作9没有子操作。

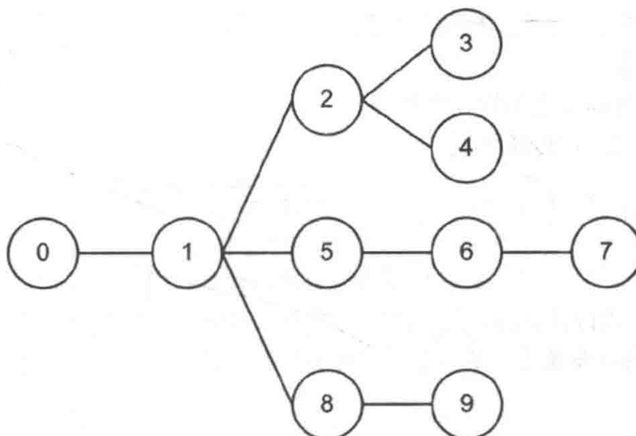


图10-2 执行计划操作之间的父-子关系

了解父-子关系对于理解执行计划执行各个操作的顺序十分关键。实际上，为了完成它们的任务，父操作需要由它们的子操作提供的数据。因此，虽然执行是从树的根部开始的，第一个被完全执行的操作是没有子操作的那个，所以，是树的叶节点。为验证这一点，我们看一下接下来的这个执行计划：

```
-------------------------------------------
| Id  | Operation                   | Name |
-------------------------------------------
|   0 | SELECT STATEMENT            |      |
|   1 |  SORT ORDER BY              |      |
|   2 |   TABLE ACCESS BY INDEX ROWID| T    |
|   3 |    INDEX RANGE SCAN         | T_PK |
-------------------------------------------
```

操作按以下顺序执行。

(1) 执行计划的入口点是操作0，它是树的根操作。但是，操作0是一个SELECT语句，没有数据可供操作。因此，它必须调用它的子操作(1)。

(2) 操作1是一个排序操作，没有数据可供操作。因此，它必须调用它的子操作(2)。

(3) 操作2是一个表扫描，需要rowid来访问t表。因此，它必须调用它的子操作(3)。

(4) 操作3是一个索引扫描，不需要来自其他操作的数据（它没有子操作）。因此，它在t_pk索引上执行索引范围扫描并将它找到的rowid传递给父操作(2)。

(5) 操作2使用从它的子操作(3)接收的rowid列表去访问t表。然后，将结果数据传递给它的父操作(1)。

(6) 操作1对它的子操作(2)传递过来的数据进行排序，然后将排序后的数据传递给它的父操作(0)。

(7) 操作0将从它的子操作(1)接收的数据传递给调用者。

注意 尽管第一个被执行的操作永远是树的根操作，但是父操作（上面的例子中是三个）可能除了调用子操作以外什么都不做。所以，为简单起见，我通常会说执行是从第一个做实际工作的操作（上面的例子中是操作3）开始的。

下面的三条通用规则总结了刚刚描述的行为。

- 父操作调用子操作。
- 子操作在它们的父操作之前被完全执行。
- 子操作向它们的父操作传递数据。

10.3.2 操作的类型

有几百种不同的操作。当然了，要完全理解一个执行计划，你应该知道每一个操作都是用来做什么的。出于我们完成整个执行计划的目的，你只需要考虑四种主要类型的操作：独立操作、迭代操作、无关联组合操作以及关联组合操作。基本上，每种类型都有特定的行为，而了解这种行为就足够阅读执行计划了。

警告　我是在2007年编写关于查询优化器的演示文稿时，提出了此处使用的四种操作类型的术语。别指望能在其他地方找到这些术语。

除了这四种类型之外，还可以将操作分为阻塞操作和非阻塞操作。简单来说，阻塞操作批量处理数据，非阻塞操作逐行处理数据。举例来说，排序操作是阻塞的，因为只有当所有输入行都被完全处理（排序）后才能返回输出的行，因为第一个输出行可能出现在输入数据集的任何地方。而另一方面，应用简单限制条件的过滤器是非阻塞的，因为它单独验证每一行。很显然对于阻塞操作，必须将数据缓存到内存中（PGA）或磁盘上（临时表空间）。为简单起见，在完成一个执行计划时，你可以认为所有的操作都是阻塞操作。但是记住，大多数的操作实际上是非阻塞的，而且出于明显的原因，SQL引擎会尝试尽可能地避免缓存数据。

10.3.3 独立操作

我将最多拥有一个子操作的所有非迭代操作（迭代操作将在下一节介绍）视为独立操作。大部分操作都是独立的。这使得执行计划的解释变得更容易，因为只有不到24种操作不属于这种类型 。控制独立操作运行的规则除了10.3.1节中描述的规则之外，还有下面这条规则：

- 一个子操作最多被执行一次。

下面是一个查询和它的执行计划的例子，基于stand-alone.sql脚本生成的输出（图10-3提供了关于它的父–子关系的图形表示）：

```
SELECT deptno, count(*)
FROM emp
WHERE job = 'CLERK' AND sal < 1200
GROUP BY deptno

-------------------------------------------------------------------
| Id  | Operation                    | Name    | Starts | A-Rows |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |      1 |      2 |
|   1 |  HASH GROUP BY               |         |      1 |      2 |
|*  2 |   TABLE ACCESS BY INDEX ROWID| EMP     |      1 |      3 |
```

```
|* 3 |    INDEX RANGE SCAN          | EMP_JOB_I |      1 |      4 |
-------------------------------------------------------------------

   2 - filter("SAL"<1200)
   3 - access("JOB"='CLERK')
```

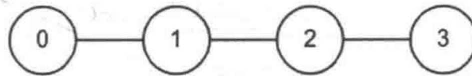


图10-3　独立操作之间的父–子关系

这个执行计划仅由独立操作组成。通过应用之前描述的规则，你会发现执行计划按照以下方式执行操作。

(1) 操作0、操作1和操作2都有一个单独的子操作（分别是1、2和3）；它们不可能是最先执行的操作。因此，执行从操作3开始。

(2) 操作3通过应用"JOB"='CLERK'访问谓词来扫描emp_job_i索引。这样做时，它从索引上抽取四个rowid（此信息在A-Rows列中给出）并将它们传递给它的父操作(2)。

(3) 操作2通过从操作3传递过来的四个rowid访问emp表。对于每个rowid，读取一行数据。接下来，它应用"SAL"<1200过滤谓词。这个过滤器会排除掉一条数据。余下的三条数据传递给它的父操作(1)。

(4) 操作1在操作2传递过来的数据上执行一个GROUP BY操作。结果集减少到两条数据并传递给它的父操作（0）。

(5) 操作0将数据发送给调用者。

注意Starts列是如何清晰地展示每一个操作都执行了一次的。

其中一条规则声明子操作在父操作之前被完整地执行。这大体上没错，但是当智能优化被引进来时情况就有些不一样了。可能发生的情况是，父操作判断完全执行子操作没有意义甚至根本不需要执行它。换句话说，父操作控制子操作的执行。我们来看两个常见的案例。注意，两个例子都摘自stand-alone.sql脚本生成的输出。

1. COUNT STOPKEY操作的优化

COUNT STOPKEY操作通常用于执行top-n查询。它的目标是一旦所需数据已经返回给了调用者就会停止处理。举例来说，下面查询的目的是只返回在emp表中找到的前10条数据：

```
SELECT *
FROM emp
WHERE rownum <= 10
```

```
---------------------------------------------------------
| Id  | Operation           | Name | Starts | A-Rows |
---------------------------------------------------------
|   0 | SELECT STATEMENT    |      |      1 |     10 |
|*  1 |  COUNT STOPKEY      |      |      1 |     10 |
|   2 |   TABLE ACCESS FULL| EMP  |      1 |     10 |
---------------------------------------------------------
```

```
1 - filter(ROWNUM<=10)
```

在这个执行计划中需要重点关注的是由操作2返回的行数被限制为10。即使操作2是对一个包含超过10条数据（实际上这张表包含14条数据）的表进行全表扫描也是这样的。结果当必要的行数被处理完毕后操作1就停止了操作2的处理工作。但是要小心，因为阻塞操作是无法停止的。事实上，必须在它们向父操作返回数据之前完全处理它们。例如，在下面的查询中，因为ORDER BY子句，会读取emp表的所有行（14）：

```
SELECT *
FROM (
  SELECT *
  FROM emp
  ORDER BY sal DESC
)
WHERE rownum <= 10

-----------------------------------------------------------
| Id  | Operation               | Name | Starts | A-Rows |
-----------------------------------------------------------
|   0 | SELECT STATEMENT        |      |      1 |     10 |
|*  1 |  COUNT STOPKEY          |      |      1 |     10 |
|   2 |   VIEW                  |      |      1 |     10 |
|*  3 |    SORT ORDER BY STOPKEY|      |      1 |     10 |
|   4 |     TABLE ACCESS FULL   | EMP  |      1 |     14 |
-----------------------------------------------------------

1 - filter(ROWNUM<=10)
3 - filter(ROWNUM<=10)
```

2. FILTER操作的优化

FILTER操作不仅会在它的子操作向它传递数据的时候应用过滤条件，此外，它也能决定完全避免子操作以及所有依赖的操作（孙子操作等）的执行。例如，在下面的查询中，从操作1处应用的过滤谓词检查绑定变量的值是否会导致空的结果集。实际上，查询只有在满足：SAL_MIN<=:SAL_MAX过滤谓词的情况下才会返回数据：

```
SELECT *
FROM emp
WHERE sal BETWEEN :sal_min AND :sal_max

------------------------------------------------------
| Id  | Operation          | Name | Starts | A-Rows |
------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |      0 |
|*  1 |  FILTER            |      |      1 |      0 |
|*  2 |   TABLE ACCESS FULL| EMP  |      0 |      0 |
------------------------------------------------------

    1 - filter(:SAL_MIN<=:SAL_MAX)
    2 - filter(("SAL"<=:SAL_MAX AND "SAL">=:SAL_MIN))
```

根据之前描述的规则，操作2应该是展示的执行计划中第一个被完全执行的操作。在现实中，查

看Starts列后可以知道只有操作0和操作1被执行了。优化简单地避免了操作2的处理，因为数据无论如何也没有机会通过操作1应用的过滤条件。

10.3.4 迭代操作

我将所有最多拥有一个可以多次执行的子操作的操作都视为迭代操作。你可以认为它们是在执行计划中实现了某种循环的操作。INLIST ITERATOR和大部分拥有PARTITION前缀的操作（例如，PARTITION RANGE ITERATOR，关于这些操作的详细描述请参见第13章）都是这种类型的操作。控制迭代操作运行的规则除了之前在10.3.1节中描述的规则之外，还有下面这条规则：

- 子操作可能会执行多次，也可能根本不执行。

下面是来自iterative.sql脚本输出的查询及其执行计划的例子：

```
SELECT *
FROM emp
WHERE job IN ('CLERK', 'ANALYST')

--------------------------------------------------------------------
| Id  | Operation                    | Name     | Starts | A-Rows |
--------------------------------------------------------------------
|   0 | SELECT STATEMENT             |          |      1 |      6 |
|   1 |  INLIST ITERATOR             |          |      1 |      6 |
|   2 |   TABLE ACCESS BY INDEX ROWID| EMP      |      2 |      6 |
|*  3 |    INDEX RANGE SCAN          | EMP_JOB_I |     2 |      6 |
--------------------------------------------------------------------

3 - access(("JOB"='ANALYST' OR "JOB"='CLERK'))
```

这个执行计划与之前在独立操作中讨论的那个类似。唯一的区别是执行计划的一部分，因为INLIST ITERATOR操作的缘故，可以被执行多次。明确地说，迭代操作的子操作可以被执行多次。在本例中，操作2和3为IN条件中的每个不同值都执行了一次。

10.3.5 无关联组合操作

我将拥有多个可以独立执行的子操作的所有操作都称为无关联组合操作。以下这些操作都属于这种类型：AND-EQUAL、BITMAP AND、BITMAP OR、BITMAP MINUS、CONCATENATION、CONNECT BY WITHOUT FILTERING、HASH JOIN、INTERSECTION、MERGE JOIN、MINUS、MULTI-TABLE INSERT、SQL MODEL、TEMP TABLE TRANSFORMATION以及UNION-ALL。控制无关联操作的规则除了10.3.1节中描述的规则之外，还包括以下两条规则。

- 子操作顺序执行，从拥有最小ID的操作开始直到拥有最大ID的操作。在开始处理随后的子操作之前，必须完全执行当前的子操作。
- 一个子操作至多执行一次并且独立于其他所有的子操作。

注意 也有特别的情况，就是MERGE JOIN操作的子操作并非严格按照刚刚提到的两个规则执行。14.3节会提供相关特殊情况的具体信息。

下面是基于unrelated-combine.sql脚本生成输出的样例查询及其执行计划（其父-子关系见图10-4）：

```
SELECT ename FROM emp
UNION ALL
SELECT dname FROM dept
UNION ALL
SELECT '%' FROM dual

-----------------------------------------------------------
| Id  | Operation          | Name | Starts | A-Rows |
-----------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |     19 |
|   1 |  UNION-ALL         |      |      1 |     19 |
|   2 |   TABLE ACCESS FULL| EMP  |      1 |     14 |
|   3 |   TABLE ACCESS FULL| DEPT |      1 |      4 |
|   4 |   FAST DUAL        |      |      1 |      1 |
-----------------------------------------------------------
```

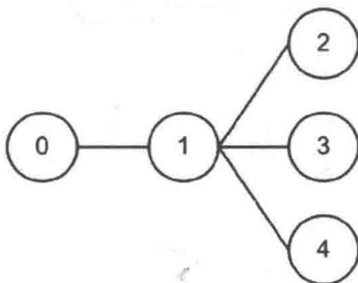


图10-4　UNION-ALL无关联组合操作的父-子关系

在这个执行计划中，无关联组合操作是UNION-ALL。其他三个是独立操作。通过应用之前给出的规则，你会发现执行计划执行的操作如下所示。

(1) 操作0有一个子操作(1)。它不可能是第一个被执行的操作。

(2) 操作1有三个子操作，其中操作2是按升序排列的第一个。因此，执行从操作2开始。

(3) 操作2扫描emp表并将14行数据返回给它的父操作(1)。

(4) 完全执行操作2之后，操作3开始执行。

(5) 操作3扫描dept表并将4行数据返回给它的父操作(1)。

(6) 完全执行操作3之后，操作4开始执行。

(7) 操作4扫描dual表并将一条数据返回给它的父操作(1)。

(8) 操作1基于它从子操作接收到的所有数据构建一个单独的19行数据的结果集，并将它们返回给父操作(0)。

(9) 操作0将数据发送给调用者。

注意Starts列是如何清晰地展示每个操作只执行了一次的。

在表10-1中我提到过存在Starts列含义不同的情况。有时候这个列提供的是一个特定的内存结构被访问的次数，而不是执行的次数。正如下例演示的那样，MERGE JOIN操作可以用来展示这样的一个

案例。注意对于操作4，其Starts列的值为4。无论如何，也没有理由将数据排序四次。但其实此内存结构被访问了四次，因此才有了4这个值出现在Starts列上。对于每一行从dept表抽取出来的数据，该内存结构都被访问了一次（第14章会详细解释合并联接是如何被执行的）：

```
-----------------------------------------------------------------
| Id  | Operation            | Name | Starts | E-Rows | A-Rows |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |      1 |        |     14 |
|   1 |  MERGE JOIN          |      |      1 |     14 |     14 |
|   2 |   SORT JOIN          |      |      1 |      4 |      4 |
|   3 |    TABLE ACCESS FULL| DEPT  |      1 |      4 |      4 |
|*  4 |   SORT JOIN          |      |      4 |     14 |     14 |
|   5 |    TABLE ACCESS FULL| EMP   |      1 |     14 |     14 |
-----------------------------------------------------------------

  4 - access("E"."DEPTNO"="D"."DEPTNO")
      filter("E"."DEPTNO"="D"."DEPTNO")
```

之前列出的所有其他操作都与本节展示的UNION-ALL操作拥有相同的行为。简而言之，无关联组合操作顺序执行它的每个子操作一次。很明显，由无关联组合操作自己执行的处理也不尽相同。

10.3.6 关联组合操作

我将拥有多个子操作且其中一个子操作控制所有其他子操作的执行的所有操作称为关联组合操作。下列操作均属于这种类型：NESTED LOOPS、FILTER、UPDATE、CONNECT BY WITH FILTERING、UNION ALL（RECURSIVE WITH)以及BITMAP KEY ITERATION。控制关联组合操作运行的规则除了之前10.3.1节中描述的规则之外，还包括以下规则。

- 拥有最小ID的子操作控制其他子操作的执行。
- 子操作从拥有最小ID的操作开始执行直到拥有最大ID的操作。但是，与无关联组合操作相反，它们不是顺序执行的，而是按某种交错的方式执行。
- 只有第一个子操作至多执行一次。其他所有子操作可能会执行多次或根本不执行。

即使这种类型的操作共享相同的特性，而它们当中的每一个，在某些方面，都有自己的行为。我们来看一下它们中各自的样例（除了BITMAP KEY ITERATION，这会在第14章中提及）。注意接下来的部分提供的所有例子都是related-combine.sql脚本生成输出的摘录。

1. NESTED LOOPS操作

这个操作用于联接两组数据。因此，它总是有两个子操作，不能多也不能少。拥有最小ID的子操作被称为外循环或驱动行源。第二个操作被称为内循环。这个操作的特性是，外循环每返回一条数据，内循环都要执行一次（第14章会详细解释嵌套循环联接是如何执行的）。

下面的查询及其执行计划就是这样的例子（图10-5展示了它的父–子关系的图形表示）：

```
SELECT *
FROM emp, dept
WHERE emp.deptno = dept.deptno
AND emp.comm IS NULL
AND dept.dname != 'SALES'
```

```
-----------------------------------------------------------------
| Id  | Operation                    | Name    | Starts | A-Rows |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |      1 |      8 |
|   1 |  NESTED LOOPS                |         |      1 |      8 |
|*  2 |   TABLE ACCESS FULL          | EMP     |      1 |     10 |
|*  3 |   TABLE ACCESS BY INDEX ROWID| DEPT    |     10 |      8 |
|*  4 |    INDEX UNIQUE SCAN         | DEPT_PK |     10 |     10 |
-----------------------------------------------------------------

  2 - filter("EMP"."COMM" IS NULL)
  3 - filter("DEPT"."DNAME"<>'SALES')
  4 - access("EMP"."DEPTNO"="DEPT"."DEPTNO")
```

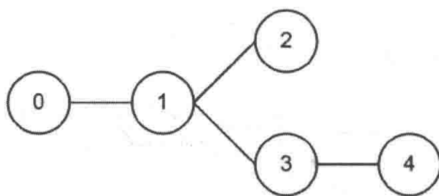


图10-5　NESTED LOOPS操作的父-子关系

在此执行计划中，NESTED LOOPS操作的两个子操作都是独立操作。通过应用之前描述的规则，你会发现执行计划按以下顺序执行各个操作。

(1) 操作0有一个子操作(1)。它不可能是第一个执行的操作。

(2) 操作1有两个子操作(2)和(3)，其中操作2是按升序排列的第一个。因此，操作2（外循环）是第一个被执行的操作。

(3) 操作2扫描emp表，应用"EMP"."COMM" IS NULL过滤谓词并将10行数据传递给它的父操作(1)。

(4) 对于操作2返回的每一条数据，NESTED LOOPS操作的第二个子操作，即内循环，都要执行一次。这是通过对比操作2的A-Rows列和操作3、操作4的Starts列确认的。

(5) 内循环由两个独立的操作构成。根据应用于这种类型的操作的规则，操作4是在操作3之前被执行的。

(6) 操作4通过应用"EMP"."DEPTNO"= "DEPT"."DEPTNO"访问谓词来扫描dept_pk索引。这样做，它通过10次执行从索引上抽取10个rowid并传递给它的父操作(3)。

(7) 操作3通过这10个从操作4返回的rowid访问dept表。对于每个rowid，都读取一行数据。接下来它应用"DEPT"."DNAME"<>'SALES'过滤谓词。这个过滤器导致两行数据被排除掉。它将剩余的8条数据传递给它的父操作(1)。

(8) 操作1将这8条数据传递给它的父操作(0)。

(9) 操作0将数据发送给调用者。

2. FILTER操作

这个操作的特性是支持不同数量的子操作。如果它拥有一个单独的子操作，就可以将它视为一个独立操作。如果它拥有两个或更多的子操作，则其功能与NESTED LOOPS操作类似。第一个子操作驱动

其他子操作的执行。

为了说明这一点，我们来看下面的查询及其执行计划（图10-6展示了其父-子关系的图形表示）：

```
SELECT *
FROM emp
WHERE NOT EXISTS (SELECT 0
                  FROM dept
                  WHERE dept.dname = 'SALES' AND dept.deptno = emp.deptno)
AND NOT EXISTS (SELECT 0
                FROM bonus
                WHERE bonus.ename = emp.ename)

-----------------------------------------------------------------
| Id  | Operation                    | Name    | Starts | A-Rows |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |      1 |      8 |
|*  1 |  FILTER                      |         |      1 |      8 |
|   2 |   TABLE ACCESS FULL          | EMP     |      1 |     14 |
|*  3 |   TABLE ACCESS BY INDEX ROWID| DEPT    |      3 |      1 |
|*  4 |    INDEX UNIQUE SCAN         | DEPT_PK |      3 |      3 |
|*  5 |   TABLE ACCESS FULL          | BONUS   |      8 |      0 |
-----------------------------------------------------------------

1 - filter( NOT EXISTS (SELECT 0 FROM "DEPT" "DEPT" WHERE "DEPT"."DEPTNO"=:B1
           AND "DEPT"."DNAME"='SALES') AND NOT EXISTS (SELECT 0 FROM "BONUS"
           "BONUS" WHERE "BONUS"."ENAME"=:B2))
3 - filter("DEPT"."DNAME"='SALES')
4 - access("DEPT"."DEPTNO"=:B1)
5 - filter("BONUS"."ENAME"=:B1)
```

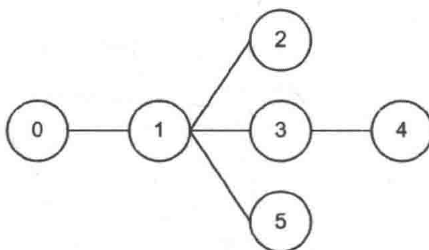


图10-6 FILTER操作的父-子关系

警告 dbms_xplan包中的display_cursor函数有时候会显示错误的谓词。然而，问题并不在于程序包。实际上是由显示错误信息的v$sql_plan和v$sql_plan_statistics_all视图引起的。在这种情况下，EXPLAIN PLAN为上面显示的计划显示正确的谓词，但是视图为操作1显示了一个错误的谓词：

```
1 - filter (( IS NULL AND IS NULL))
```

注意，根据Oracle的说法，这不是一个bug，只是当前实现的一个限制。

在这个执行计划中，FILTER操作的三个子操作为独立操作。应用之前描述的规则，你可以发现执行计划按以方式执行各个操作。

(1) 操作0有一个子操作(1)。它不可能是第一个被执行的操作。

(2) 操作1有三个子操作（2、3和5），操作2是它们当中按升序排列的第一个。因此，执行从操作2开始。

(3) 操作2扫描emp表并将14条数据返回给它的父操作(1)。

(4) 对于操作2返回的每条数据，FILTER操作的第二个和第三个子操作都应该执行一次。而实际上，某种缓存被实现以将执行减至最少。这是通过将操作2的A-Rows列与操作3、5的Starts列相对比得知的。操作3被执行了三次，为emp表的deptno列的每个不重复值执行了一次。操作5执行了八次，为emp表在应用完由操作3施加的过滤器之后的ename列的每个不重复值执行了一次。下面的查询表明starts列的数值和不重复值的数量相匹配：

```
SQL> SELECT deptno, dname, count(*)
  2  FROM emp NATURAL JOIN dept
  3  GROUP BY deptno, dname;

DEPTNO DNAME        COUNT(*)
------ ---------- --------
    10 ACCOUNTING        3
    20 RESEARCH          5
    30 SALES             6
```

(5) 根据独立操作的规则，操作4是在操作3之前执行的，通过应用"DEPT"."DEPTNO"=:B1访问谓词来扫描dept_pk索引。绑定变量（B1）用来传递通过子查询检查的值。通过在三次执行中都这样做，操作从索引中提取三个rowid并将它们传递给它的父操作(3)。

(6) 操作3通过从它的子操作(4) 传递过来的rowid访问dept表并应用"DEPT"."DNAME"='SALES'过滤谓词。因为这个操作只是用来应用一个限制条件，它不向父操作(1) 返回任何数据。它仅通知父操作条件是否满足。无论如何，应该注意到只找到一行满足过滤谓词的数据。因为使用了NOT EXISTS，这个匹配的行被丢弃掉了。

(7) 操作5扫描bonus表并应用"BONUS"."ENAME"=:B1过滤谓词。绑定变量（B1）用来传递通过子查询检查的值。因为这个操作只用于应用一个限制条件，它不向其父操作(1) 返回任何数据。但是要注意没有找到满足过滤谓词的数据。因为使用了NOT EXISTS，没有数据被丢弃掉。

(8) 在应用完由操作3和操作5实现的过滤谓词后，操作1将结果数据返回给它的父操作（0）。

(9) 操作0将数据发送给调用者。

3. UPDATE操作

这个操作是执行某个UPDATE语句时使用的。它的特性是支持不同数量的子操作。大多数时候，它拥有一个单独的子操作而且因此被认为是独立操作。只有在SET子句中使用子查询时，才会有两个或更多的子操作可用。如果它拥有不止一个子操作，那么第一个子操作驱动其他子操作的执行。

下面是一个样例SQL语句及其执行计划（图10-7展现了它的父-子关系的图形表示）：

```
UPDATE emp e1
SET sal = (SELECT avg(sal) FROM emp e2 WHERE e2.deptno = e1.deptno),
    comm = (SELECT avg(comm) FROM emp e3)
--------------------------------------------------------------
| Id  | Operation              | Name | Starts | E-Rows | A-Rows |
--------------------------------------------------------------
```

```
|   0 | UPDATE STATEMENT    |      |     1 |      |     0 |
|   1 |  UPDATE             | EMP  |     1 |      |     0 |
|   2 |   TABLE ACCESS FULL | EMP  |     1 |   14 |    14 |
|   3 |   SORT AGGREGATE    |      |     3 |    1 |     3 |
|*  4 |    TABLE ACCESS FULL| EMP  |     3 |    5 |    14 |
|   5 |   SORT AGGREGATE    |      |     1 |    1 |     1 |
|   6 |    TABLE ACCESS FULL| EMP  |     1 |   14 |    14 |
-------------------------------------------------------------
```

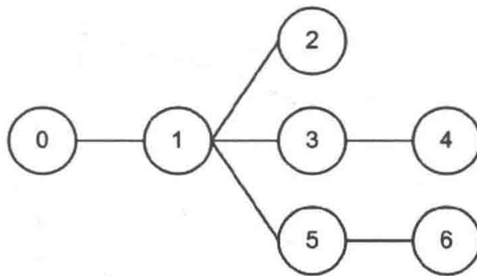


图10-7 UPDATE操作的父–子关系

```
   4 - filter("E2"."DEPTNO"=:B1)
```

在这个执行计划中，UPDATE关联组合操作的全部三个子操作都是独立操作。之前描述的规则表明执行计划按以下方式执行各个操作。

(1) 操作0有一个子操作(1)。它不可能是第一个被执行的操作。

(2) 操作1有三个子操作（2、3和5），且操作2是这三个中按升序排列的第一个。因此，执行从操作2开始。

(3) 操作2扫描emp表并向它的父操作(1)返回14行数据。

(4) 第二个和第三个子操作（3和5）可能会被执行多次（最多会与操作2返回的行数相等）。因为这些操作都是独立的，且每个操作都有一个子操作，它们的执行从子操作（4和6）开始。

(5) 对于由操作2返回的deptno列中的每个不重复值，操作4扫描emp表并应用"E2"."DEPTNO"=:B1过滤谓词。通过在三次执行中这么做，操作提取出14行数据并将它们传递给它的父操作(3)。

(6) 操作3计算从操作4传递给它的数据的平均工资，并将结果返回给它的父操作(1)。

(7) 操作6扫描emp表，提取14行数据，并将它们传递给它的父操作(5)。注意这个子查询只执行了一次，因为它并不与主查询相互关联。

(8) 操作5计算从操作6传递给它的数据的平均佣金，并将结果返回给它的父操作(1)。

(9) 操作1使用它的子操作（3和5）返回的值来更新由操作2传递过来的每一行数据，并向它的父操作(0)传递更新的行数。注意，即使UPDATE语句修改了这14行数据，这个操作的A-Rows列仍显示为0。

(10) 操作0向调用者发送被修改的行数。

4. CONNECT BY WITH FILTERING操作

这个操作是用来处理层次查询的。它的特征是有两个子操作。第一个用来获取层次的顶级，第二个为层次中的每一个级别都执行一次。

下面是一个样例查询及其计划（图10-8展示了它的父–子关系的图形表示）。注意，该执行计划是

在11.2版本下生成的（原因在之前解释过了）：

```
SELECT level, rpad('-',level-1,'-')||ename AS ename, prior ename AS manager
FROM emp
START WITH mgr IS NULL
CONNECT BY PRIOR empno = mgr

--------------------------------------------------------------------
| Id  | Operation                     | Name       | Starts | A-Rows |
--------------------------------------------------------------------
|   0 | SELECT STATEMENT              |            |      1 |     14 |
|*  1 |  CONNECT BY WITH FILTERING    |            |      1 |     14 |
|*  2 |   TABLE ACCESS FULL           | EMP        |      1 |      1 |
|   3 |   NESTED LOOPS                |            |      4 |     13 |
|   4 |    CONNECT BY PUMP            |            |      4 |     14 |
|   5 |    TABLE ACCESS BY INDEX ROWID| EMP        |     14 |     13 |
|*  6 |     INDEX RANGE SCAN          | EMP_MGR_I  |     14 |     13 |
--------------------------------------------------------------------

   1 - access("MGR"=PRIOR "EMPNO")
   2 - filter("MGR" IS NULL)
   6 - access("connect$_by$_pump$_002"."PRIOR empno"="MGR")
       filter("MGR" IS NOT NULL)
```

警告　上面的查询代表了v$sql_plan和v$sql_plan_statistics_all视图给出错误信息的另一种情况。在本例中，EXPLAIN PLAN显示了上面显示的正确谓词，错误显示的谓词是与操作1关联的那个：

```
1 - access ( "MGR"=PRIOR NULL)
```

此外，与操作6关联的访问谓词在11.1及之前的版本中都是错误的。

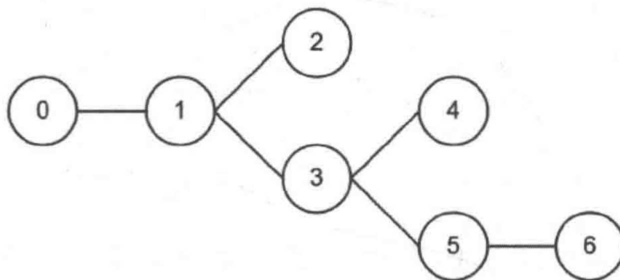


图10-8　CONNECT BY WITH FILTERING操作的父–子关系

在这个执行计划中，CONNECT BY WITH FILTERING操作的第一个子操作是独立操作。不同的是，第二个子操作本身是一个关联组合操作。在这种情况下读取一个执行计划，你只需要简单地沿着关系树下行递归应用规则。

为了帮助你更容易地理解层次查询的执行计划，可以查看一下查询返回的数据：

```
LEVEL ENAME      MANAGER
```

```
----- -------- -------
    1 KING
    2 -JONES   KING
    3 --SCOTT  JONES
    4 ---ADAMS SCOTT
    3 --FORD   JONES
    4 ---SMITH FORD
    2 -BLAKE   KING
    3 --ALLEN  BLAKE
    3 --WARD   BLAKE
    3 --MARTIN BLAKE
    3 --TURNER BLAKE
    3 --JAMES  BLAKE
    2 -CLARK   KING
    3 --MILLER CLARK
```

应用早前描述的规则，你会发现执行计划按以下方式执行各个操作。

(1) 操作0有一个子操作(1)。它不可能是第一个被执行的操作。

(2) 操作1有两个子操作(2和3)，而且按升序排列时操作2排在第一。因此，执行从操作2开始。

(3) 操作2扫描emp表，应用"MGR" IS NULL过滤谓词，然后将层次的根（KING）返回给它的父操作(1)。

(4) 操作3是操作1的第二个子操作。因此它会为层次中的每一个级别都执行，在本例中执行四次。当然，之前讨论的关于NESTED LOOPS操作的规则适用于操作3。第一个子操作(4)被执行了，对于它返回的每一行，会将内循环（由操作5和6操作组成）都执行一次。注意，正如预期的，操作4的A-Rows列与操作5和6的Starts列之间是匹配的。

(5) 对于第一次执行，操作4通过CONNECT BY PUMP操作获取层级的根。在本例中，级别1只有一条数据（KING）。通过这个值，操作6通过应用"MGR"=PRIOR "EMPNO"访问谓词（显示为"connect$_by$_pump$_002"."PRIOR empno"="MGR"）对emp_mgr_i索引做扫描，应用过滤谓词"MGR" IS NOT NULL，提取出rowid，然后将它们返回给它的父操作(5)。操作5通过这些rowid访问emp表，并将数据返回给它的父操作(3)。

(6) 对于操作4的第二次执行，做的每件事都与第一次执行的一样。唯一的不同是来自级别2的数据（JONES、BLAKE以及CLARK）被传递给操作4用于处理（一个接一个，每一行都会引发操作4的启动）。

(7) 对于操作4的第三次执行，做的每件事都与第一次执行的一样。唯一的不同是来自级别3的数据（SCOTT、FORD、ALLEN、WARD、MARTIN、TURNER、JAMES以及MILLER）被传递给操作4用于处理。

(8) 对于操作4的第四次和最后一次执行，做的每件事都与第一次执行的一样。唯一的不同是来自级别4的数据（ADAMS和SMITH）被传递给操作4用于处理。

(9) 操作3获取从它的子操作传递过来的数据，然后将它们返回给它的父操作(1)。

(10) 操作1应用"MGR" IS NOT NULL过滤谓词。

(11) 操作0将数据发送给调用者。

在10.2.0.3及之前的版本中，生成的执行计划有着细微的不同。如下例所示，CONNECT BY WITH FILTERING操作有第三个子操作（操作8）。然而，在本例中，并没有执行它。操作8的Starts列中的值证实了这一点。实际上，仅当CONNECT BY WITH FILTERING操作使用临时表空间时，才会执行第三个子

操作。到那时候性能恐怕会严重下降。在10.2.0.4及之后的版本中已修复这个问题，这个问题被称为bug 5065418：

```
-----------------------------------------------------------------------
| Id  | Operation                     | Name       | Starts |  A-Rows |
-----------------------------------------------------------------------
|*  1 | CONNECT BY WITH FILTERING     |            |      1 |      14 |
|*  2 |  TABLE ACCESS FULL            | EMP        |      1 |       1 |
|   3 |  NESTED LOOPS                 |            |      4 |      13 |
|   4 |   BUFFER SORT                 |            |      4 |      14 |
|   5 |    CONNECT BY PUMP            |            |      4 |      14 |
|   6 |   TABLE ACCESS BY INDEX ROWID| EMP        |     14 |      13 |
|*  7 |    INDEX RANGE SCAN           | EMP_MGR_I  |     14 |      13 |
|   8 |   TABLE ACCESS FULL           | EMP        |      0 |       0 |
-----------------------------------------------------------------------
```

5. UNION ALL (RECURSIVE WITH)操作

UNION ALL (RECURSIVE WITH)操作从11.2版本开始可用。添加它是为了实现递归子查询因子子句。因此，会将它用于层次查询。注意实际上存在着两个有关的操作：

```
UNION ALL (RECURSIVE WITH)BREADTH FIRST
UNION ALL (RECURSIVE WITH)DEPTH FIRST
```

顾名思义，区别源自于你可以将搜索子句指定为BREADTH FIRST BY或者DEPTH FIRST BY。

下面是一个样例查询及其执行计划：

```
WITH
  e (xlevel, empno, ename, job, mgr, hiredate, sal, comm, deptno)
  AS (
    SELECT 1, empno, ename, job, mgr, hiredate, sal, comm, deptno
    FROM emp
    WHERE mgr IS NULL
    UNION ALL
    SELECT mgr.xlevel+1, emp.empno, emp.ename, emp.job, emp.mgr, emp.hiredate, emp.sal,
    FROM emp, e mgr
    WHERE emp.mgr = mgr.empno
  )
SELECT *
FROM e

---------------------------------------------------------------------------------
| Id  | Operation                                 | Name       | Starts | A-Rows |
---------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                          |            |      1 |     14 |
|   1 |  VIEW                                     |            |      1 |     14 |
|   2 |   UNION ALL (RECURSIVE WITH)BREADTH FIRST |            |      1 |     14 |
|*  3 |    TABLE ACCESS FULL                      | EMP        |      1 |      1 |
|   4 |    NESTED LOOPS                           |            |      4 |     13 |
|   5 |     NESTED LOOPS                          |            |      4 |     13 |
|   6 |      RECURSIVE WITH PUMP                  |            |      4 |     14 |
|*  7 |      INDEX RANGE SCAN                     | EMP_MGR_I  |     14 |     13 |
|   8 |     TABLE ACCESS BY INDEX ROWID           | EMP        |     13 |     13 |
---------------------------------------------------------------------------------

  3 - filter("MGR" IS NULL)
```

```
7 - access("EMP"."MGR"="MGR"."EMPNO")
    filter("EMP"."MGR" IS NOT NULL)
```

读取一个包含UNION ALL （RECURSIVE WITH)操作的执行计划与读取一个包含CONNECT BY WITH FILTERING操作的执行计划没什么两样。事实上，两个操作的用途基本上是相同的。只是要注意执行计划中使用的PUMP操作不一样。在前者中它被称为RECURSIVE WITH PUMP，在后者中它被称为CONNECT BY PUMP。无论如何，这种差别，对于读取执行计划的目的来说，是无关紧要的。

10.3.7 分而治之

在前面的章节中，你了解了如何读取由三种类型的操作组成的执行计划。到目前为止你看到的执行计划都十分简单（较短）。然而，多半情况下，你需要面对复杂（较长）的执行计划。这不是因为大部分的SQL语句都是复杂的，而是因为很可能简单的SQL语句能被查询优化器很好地优化，因此你永远不必去怀疑简单语句的性能。

要认识到读取长的执行计划与读取短的执行计划没有本质区别。你所需要做的仅是有条理地应用在之前章节中提供的规则。有了它们，就无所谓执行计划有多少行。只要按相同的方式进行就可以了。

为了向你展示如何处理行数稍多的执行计划，我们来看一下图10-9中的执行计划所执行的操作（图10-10展示了其父-子关系的图形表示）。我有意不提供用来生成它的SQL语句。对于我们的目的而言，你不需要关心SQL语句本身。换句话说，执行计划才是关键。

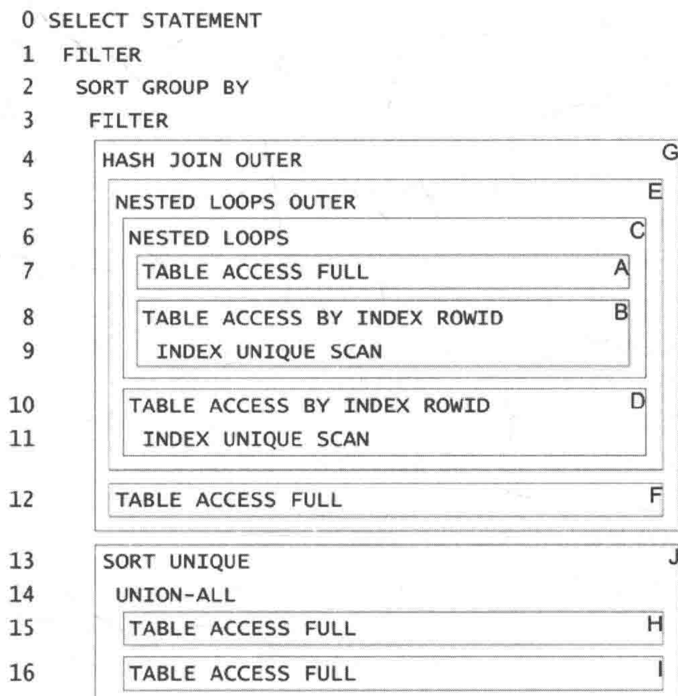


图10-9 一个执行计划按块进行分解。左边的数字用于识别操作。右边的字母用于识别块

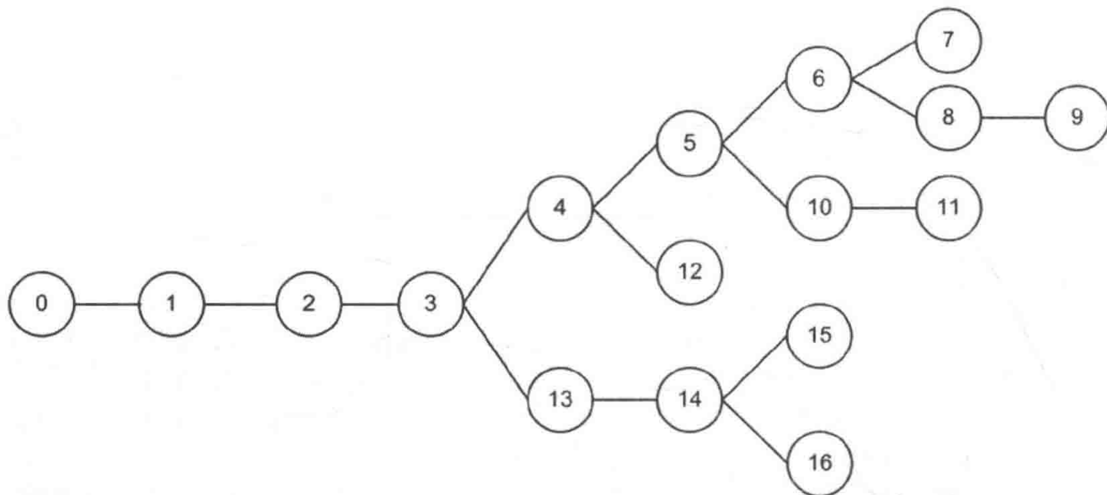


图10-10　图10-9中展示的执行计划的父–子关系

首先，有必要将查询计划分解为基础的块，并识别执行的顺序。为此，你需要实施以下步骤。最开始为了读取执行计划，你必须识别组成它的组合操作（包括关联的和无关联的）。也就是说，你需要识别每一个拥有不止一个子操作的操作。在图10-9所示的例子中，组合操作包括：3、4、5、6和14。然后，对于每个组合操作中的每个子操作，都定义一个块。因为在图10-9中有五个组合操作，而且它们当中的每一个都有两个子操作，所以一共有十个块。例如，对于操作3，第一个子操作包含从第4行到第12行（块G），而第二个子操作包含从第13行到第16行（块J）。注意在图10-9中，每个块都被一个方框分隔开来。最终，你需要找出这些块的执行顺序。为了观察这是如何完成的，我们完成图10-9中展示的执行计划，并应用之前讨论的规则。

(1) 操作0是一个独立操作，它的子操作(1)在它之前执行。

(2) 操作1是一个独立操作，它的子操作(2)在它之前执行。

(3) 操作2是一个独立操作，它的子操作(3)在它之前执行。

(4) 操作3是一个独立操作，它的子操作在它之前执行。因为第一个子块（G）在第二个子块（J）之前执行，我们继续看第一个子块的第一个操作(4)。

(5) 操作4是一个无关联组合操作，它的子操作在它之前执行。因为第一个子块（E）在第二个子块（F）之前执行，我们继续看第一个子块（E）的第一个操作(5)。

(6) 操作5是一个关联组合操作，它的子操作在它之前执行。因为第一个子块（C）在第二个子块（D）之前执行，我们继续看第一个子块（C）的第一个操作(6)。

(7) 操作6是一个关联组合操作，它的子操作在它之前执行。因为第一个子块（A）在第二个子块（B）之前执行，我们继续看第一个子块（A）的第一个操作(7)。

(8) 操作7是一个独立操作而且没有子操作。这意味着你终于找到了第一个被执行的操作（因为它在块A中）。该操作扫描一张表，并将数据返回给它的父操作(6)。

(9) 块B需要为由块A返回的每一行都执行一遍。在这个块中，起初操作9扫描一个索引，然后操作8使用返回的rowid访问一张表，并最终将数据返回给它的父操作(6)。

(10) 操作6在由块A和B返回的数据之间执行联接操作，然后将结果返回给它的父操作(5)。

(11) 块D需要为由块C返回的每一行都执行一遍。换句话说，对于由操作6返回给其父操作(5) 的每一行，它都被执行了一次。在这个块中，一开始是操作11扫描一个索引。然后，操作10通过返回的rowid访问一张表，并将数据返回给它的父操作(5)。

(12) 操作5在由块C和D返回的数据之间执行联接操作，然后将结果返回给它的父操作(4)。

(13) 操作12（块F）仅执行一次。它扫描一张表，然后将结果返回给它的父操作(4)。

(14) 操作4在由块E和F返回的数据之间执行联接操作，然后将结果返回给它的父操作(3)。

(15) 块J基本上是对于块G返回的每一行数据都要执行一次。换句话说，对于由操作4返回给其父操作(3) 的每一行数据，它都要被执行一次。在这个块中，首先操作15扫描一张表，然后将数据返回给它的父操作(14)。接下来，操作16扫描一张表，并将数据返回给它的父操作(14)。做完这些，操作14将它的各个子操作返回的数据放到一起，并将结果返回给它的父操作(13)。最后，操作13移除部分冗余的数据。注意，这个块不会将数据返回给其父操作。实际上，父操作是一个FILTER操作，而且第二个子操作仅用来应用一个限制条件。

(16) 一旦操作3在块J上应用了过滤条件，就将结果返回给它的父操作(2)。

(17) 操作2执行一个GROUP BY操作并将结果返回给它的父操作(1)。

(18) 操作1应用一个过滤条件然后将结果返回给调用者。

概括起来，注意各个块是按照它们的标识符顺序执行的（从A一直到J）。一些块（A、C、E、F以及G）至多执行一次，而其他的块（B、D、H、I以及J）则可能执行多次（或根本不执行），这要取决于驱动它们的操作返回了多少条数据。

10.3.8 特殊情况

前面章节中描述的规则适用于绝大部分的执行计划。虽然如此，还是有一些特殊情况。通常可以通过观察操作获知执行计划做了哪些事情，它们应用的谓词，它们是在哪些表上执行的以及它们的运行时行为（尤其是Starts和A-Rows列）。接下来的小节介绍了从众多可能的情况中挑选出来的三个例子。注意以下例子都是对special_cases.sql脚本生成输出的摘录。

1. SELECT子句中的子查询

这个例子展示了在SELECT子句中包含一个子查询的查询语句的执行计划是什么样子的。查询及其执行计划如下所示：

```
SELECT ename, (SELECT dname
               FROM dept
               WHERE dept.deptno = emp.deptno)
FROM emp

------------------------------------------------------------------
| Id  | Operation                   | Name    | Starts | A-Rows |
------------------------------------------------------------------
|   0 | SELECT STATEMENT            |         |      1 |     14 |
|   1 |  TABLE ACCESS BY INDEX ROWID| DEPT    |      3 |      3 |
|*  2 |   INDEX UNIQUE SCAN         | DEPT_PK |      3 |      3 |
|   3 |  TABLE ACCESS FULL          | EMP     |      1 |     14 |
------------------------------------------------------------------
```

10

```
   2 - access("DEPT"."DEPTNO"=:B1)
```

奇怪的是，在这个执行计划中操作0有多个子操作。如果仔细观察Starts列，就会注意到尽管操作1和操作2被执行了三次，操作3仅被执行了一次。还要注意操作1和操作2，因为它们引用了dept表实现了子查询。这个不寻常的执行计划按以下步骤执行各个操作。

(1) 操作3，也就是第一个被执行的操作，扫描emp表并将所有的数据返回给它的父操作（0）。

(2) 对于操作3返回的每一行数据，子查询都应该被执行一次。然而，在本例中SQL引擎也缓存了结果，因此子查询只是为deptno列中的每个不重复值都执行了一次。

(3) 为执行子查询，操作2通过应用"DEPT"."DEPTNO"=:B1访问谓词来扫描dept_pk索引，提取rowid，并将它们返回给它的父操作(1)。绑定变量（B1）用来将需要检索的值传递给子查询。然后操作1使用这些rowid访问dept表，并将数据传递给它的父操作（0）。

(4) 操作0将数据发送给调用者。

2. WHERE子句中的子查询 #1

这个例子展示一个与在WHERE子句中包含着子查询的查询语句有关的特殊执行计划。该查询及其执行计划如下所示：

```
SELECT deptno
FROM dept
WHERE deptno NOT IN (SELECT deptno FROM emp)

------------------------------------------------------
| Id  | Operation          | Name    | Starts | A-Rows |
------------------------------------------------------
|   0 | SELECT STATEMENT   |         |      1 |      1 |
|*  1 |  INDEX FULL SCAN   | DEPT_PK |      1 |      1 |
|*  2 |   TABLE ACCESS FULL| EMP     |      4 |      3 |
------------------------------------------------------

   1 - filter( NOT EXISTS (SELECT 0 FROM "EMP" "EMP" WHERE
              LNNVL("DEPTNO"<>:B1)))
   2 - filter(LNNVL("DEPTNO"<>:B1))
```

警告 这个查询是v$sql_plan和v$sql_plan_statistics_all视图给出错误信息的另一个案例。在本例中，EXPLAIN PLAN显示如上所示的正确谓词。错误显示的谓词是与操作1有关的那个：

```
1- filter（IS NULL）
```

乍一看，这个执行计划是由两个独立操作组成的。如果仔细观察Starts列，会注意到某些地方有点奇怪。事实上，尽管父操作(1) 只被执行了一次，但子操作(2) 却被执行了四次。实际上，该执行计划是按照以下步骤执行各个操作的。

(1) 操作1，也就是第一个被执行的操作，扫描dept_pk索引。对于deptno列中的每个值，都会执行操作2。就像过滤谓词显示的那样，操作2应用NOT EXISTS （SELECT 0 FROM "EMP" "EMP" WHERE LNNVL（"DEPTNO"<>:B1))子查询。注意，查询优化器将NOT IN转化为了NOT EXISTS。绑定变量（B1）用来向子

查询传递需要检索的值。

(2) 操作2扫描emp表，应用LNNVL ("DEPTNO"<>:B1)过滤谓词，并将数据返回给它的父操作(1) 。

(3) 对于满足过滤谓词的每条数据，操作1都将其传递给它的父操作（0）。

(4) 操作0将数据发送给调用者。

对于同一个查询语句，查询优化器还有可能生成下面的执行计划。但是，因为它使用了一个仅从11.1版本开始可用的特性（NULL-aware anti-join），不要指望在10.2版本中看见这种类型的执行计划。（依我看来，这个执行计划远比上一个更容易读取。）

```
-----------------------------------------------------------
| Id  | Operation          | Name    | Starts | A-Rows |
-----------------------------------------------------------
|   0 | SELECT STATEMENT   |         |      1 |      1 |
|*  1 |  HASH JOIN ANTI NA |         |      1 |      1 |
|   2 |   INDEX FULL SCAN  | DEPT_PK |      1 |      4 |
|   3 |   TABLE ACCESS FULL| EMP     |      1 |     14 |
-----------------------------------------------------------

   1 - access("DEPTNO"="DEPTNO")
```

3. WHERE子句中的子查询 #2

这个例子是上一个的扩展。它也涉及在WHERE子句中的子查询。之所以展示它，是因为想提醒大家注意这样的事实：即使实现子查询的编码远比一个简单的查找复杂得多，查询优化器也能够生成类似前面小节中讨论的那种执行计划。该查询及其执行计划如下：

```
SELECT *
FROM t1
WHERE n1 = 8 AND n2 IN (SELECT t2.n1
                        FROM t2, t3
                        WHERE t2.id = t3.id AND t3.n1 = 4);
------------------------------------------------------------------
| Id  | Operation                     | Name | Starts | A-Rows |
------------------------------------------------------------------
|   0 | SELECT STATEMENT              |      |      1 |      7 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T1   |      1 |      7 |
|*  2 |   INDEX RANGE SCAN            | I1   |      1 |      7 |
|*  3 |    HASH JOIN                  |      |     13 |      1 |
|*  4 |     TABLE ACCESS FULL         | T3   |     13 |   1183 |
|*  5 |     TABLE ACCESS FULL         | T2   |     13 |    910 |
------------------------------------------------------------------

  2 - access("N1"=8)
      filter( EXISTS (SELECT /*+ PUSH_SUBQ LEADING ("T3" "T2") FULL ("T3")
             USE_HASH ("T2") FULL ("T2") */ 0 FROM "T3" "T3","T2" "T2" WHERE
             "T2"."ID"="T3"."ID" AND "T2"."N1"=:B1 AND "T3"."N1"=4))
  3 - access("T2"."ID"="T3"."ID")
  4 - filter("T3"."N1"=4)
  5 - filter("T2"."N1"=:B1)
```

警告　这个查询是v$sql_plan和v$sql_plan_statistics_all视图给出错误信息的另一个案例。在本例中，EXPLAIN PLAN显示如上所示的正确谓词。错误显示的谓词是与操作2的过滤条件相关的那个：

```
2 - access ("N1"=8)
    filter ( IS NOT NULL)
```

同样在本例中，如果仔细观察Starts列，会注意到某些地方有点奇怪。直到2操作为止都是执行了一次，而从3到5的操作却执行了13次。该执行计划按以下步骤执行各个操作。

(1) 操作2，也就是第一个被执行的操作，通过扫描i1索引来应用"N1"=8访问谓词。它从索引中提取的，对于满足访问谓词的键，不仅是rowid而且还有n2列的值。对于n2列中的每个不重复值，子查询（操作3到5）都被执行一次。这是通过应用过滤谓词来完成的。注意查询优化器将IN转换成了EXISTS。子查询实施的联接是通过一个散列联接实现的，这是一个无关联组合操作。

(2) 操作4，散列联接的第一个子操作，通过全表扫描读取t3表并将满足"T3"."N1"=4过滤谓词的数据返回给它的父操作(3)。

(3) 操作5，散列联接的第二个子操作，通过全表扫描读取t2表并将满足"T2"."N1"=:B1过滤谓词的数据返回给它的父操作(3)。绑定变量（B1）用来向子查询传递需要检索的值。

(4) 操作3联接由操作4和5传递过来的两组结果集。当至少有一行数据被找到时，它将数据返回给它的父操作(2)。

(5) 对于每条满足由子查询实现的条件的数据，操作2都向其父操作(1) 传递一个rowid。

(6) 操作1使用从其子操作(2)接收的rowid来访问t1表并提取它的各个列。它将数据传递给它的父操作(0)。

(7) 操作0将数据发送给调用者。

10.3.9　自适应执行计划

对象统计信息并不总是会为查询优化器提供用于找出最优执行计划所需的全部信息。为了改进这种情况，在解析阶段，查询优化器可以利用动态采样对要处理的数据获取额外的洞察力。（动态采样在第9章中描述过。）此外，自从12.1版本开始，查询优化器能够将某些决定推迟到执行阶段。其思路是，利用在执行计划的执行部分可以收集的信息来决定应该如何执行其他部分。基于这个目的，查询优化器引入了所谓的子计划。同时引入的还有负责决定应该激活哪个子计划的操作。

注意　自适应执行计划只有在企业版中才可用。

从12.1版本开始，查询优化器能在以下状况中使用自适应执行计划。

- 从嵌套循环联接切换到散列联接，反之亦然。
- 为并行执行的SQL语句从散列向广播切换分配方法。

与并行处理有关的案例会在第15章中介绍。下面的例子演示切换联接方法是如何实现的。这是一段来自adaptive_plan.sql脚本生成输出的摘录。此查询是一个两张表之间的简单联接。它通过普通的嵌套循环联接执行：

```
SQL> EXPLAIN PLAN FOR
  2  SELECT *
  3  FROM t1, t2
  4  WHERE t1.id = t2.id
  5  AND t1.n = 666;

SQL> SELECT * FROM table(dbms_xplan.display(format=>'basic +predicate +note'));

PLAN_TABLE_OUTPUT
----------------------------------------------
Plan hash value: 1837274416

----------------------------------------------
| Id  | Operation                    | Name  |
----------------------------------------------
|   0 | SELECT STATEMENT             |       |
|   1 |  NESTED LOOPS                |       |
|   2 |   NESTED LOOPS               |       |
|*  3 |    TABLE ACCESS FULL         | T1    |
|*  4 |    INDEX UNIQUE SCAN         | T2_PK |
|   5 |   TABLE ACCESS BY INDEX ROWID| T2    |
----------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("T1"."N"=666)
   4 - access("T1"."ID"="T2"."ID")

Note
-----
   - this is an adaptive plan
```

注意，上面摘录中末尾的Note部分指出了这个执行计划是自适应的。然而，就执行计划本身而言，却没有任何特别的地方。而事实是，默认情况下，dbms_xplan包的display函数只显示默认的执行计划。简单来说，这是查询优化器在不考虑自适应执行计划时会选择的执行计划。如果想看见包含子计划的完整执行计划，必须在使用dbms_xplan包时指定adaptive修饰符。在这种情况下，有三个额外的操作会在该执行计划中显示：

```
SQL> SELECT * FROM table(dbms_xplan.display(format=>' basic +predicate +note +adaptive'));

PLAN_TABLE_OUTPUT
-------------------------------------------------
Plan hash value: 1837274416

-------------------------------------------------
|   Id  | Operation                       | Name  |
-------------------------------------------------
|     0 | SELECT STATEMENT                |       |
|- *  1 |  HASH JOIN                      |       |
|     2 |   NESTED LOOPS                  |       |
|     3 |    NESTED LOOPS                 |       |
|-    4 |     STATISTICS COLLECTOR        |       |
```

```
| *  5 |      TABLE ACCESS FULL         | T1    |
| *  6 |     INDEX UNIQUE SCAN          | T2_PK |
|    7 |    TABLE ACCESS BY INDEX ROWID| T2    |
|-   8 |   TABLE ACCESS FULL            | T2    |
------------------------------------------------

  1 - access("T1"."ID"="T2"."ID")
  5 - filter("T1"."N"=666)
  6 - access("T1"."ID"="T2"."ID")

Note
-----
  - this is an adaptive plan (rows marked '-' are inactive)
```

这样的一个执行计划并不容易读取，因为它其实包含两个不同的执行计划。首先，是基于嵌套循环联接的默认执行计划：

```
---------------------------------------------
| Id  | Operation                    | Name  |
---------------------------------------------
|   0 | SELECT STATEMENT             |       |
|   1 |  NESTED LOOPS                |       |
|   2 |   NESTED LOOPS               |       |
|   3 |    TABLE ACCESS FULL         | T1    |
|   4 |    INDEX UNIQUE SCAN         | T2_PK |
|   5 |   TABLE ACCESS BY INDEX ROWID| T2    |
---------------------------------------------
```

接下来，是基于散列联接的自适应执行计划：

```
-----------------------------------
| Id  | Operation          | Name |
-----------------------------------
|   0 | SELECT STATEMENT   |      |
|   1 |  HASH JOIN         |      |
|   2 |   TABLE ACCESS FULL| T1   |
|   3 |   TABLE ACCESS FULL| T2   |
-----------------------------------
```

基本上，当t1表的扫描返回少量的数据时，第一个执行计划比第二个好。因此，要决定应该使用哪个执行计划，查询优化器会估算能够被嵌套循环联接有效处理的最大行数（称作转折点）。为了在执行阶段期间决定应该使用哪个执行计划，STATISTICS COLLECTOR操作缓存并记录t1表的扫描返回的记录数。然后，只有当记录数低于转折点的数值时，嵌套循环联接才会被执行。否则，散列联接会被执行。此时的执行计划通常被称作最终执行计划。一旦最终执行计划确定下来，就会禁用STATISTICS COLLECTOR操作，因此，不会发生进一步的缓存。此外，与转折点方法有关的操作也会禁用。

注意　要知道执行计划切换只会发生在子游标第一次执行时。所有后续执行都使用最终执行计划。

---

v$sql动态性能视图提供一个新的列帮助你了解，对于一个特定的子游标其最终执行计划是否已经选定。这个列就是is_resolved_adaptive_plan。它会被设置为以下值。

- NULL意味着与该游标关联的执行计划不是自适应的。
- N意味着最终执行计划还没有被确定下来。这个值只有在最终执行计划被确定下来之前才可以观察到。
- Y意味着最终执行计划已经被确定下来。

两个初始化参数控制自适应执行计划。

- optimizer_adaptive_features完全启用或禁用该特性。将这个参数设置为FALSE时，会禁用自适应执行计划。默认值是TRUE。
- optimizer_adaptive_reporting_only在报告模式下启用或禁用自适应执行计划。这个模式对于评估执行计划是否会因为自适应执行计划而改变非常有用。当设置为TRUE时，就会生成自适应执行计划，SQL引擎会检查转折点，但是SQL引擎只会使用默认的执行计划。然后，通过下面的例子所示的报告特性，可以检查如果完全启用自适应执行计划，那么会使用哪一个执行计划。默认值是FALSE：

```
SQL> ALTER SESSION SET optimizer_adaptive_reporting_only = TRUE;

SQL> SELECT *
  2  FROM t1, t2
  3  WHERE t1.id = t2.id
  4  AND t1.n = 666;

SQL> SELECT *
  2  FROM table(dbms_xplan.display_cursor(format=>'basic +predicate +note +adaptive +report'));

EXPLAINED SQL STATEMENT:
------------------------
SELECT * FROM t1, t2 WHERE t1.id = t2.id AND t1.n = 666

Plan hash value: 1837274416

-------------------------------------------------
| Id  | Operation                      | Name   |
-------------------------------------------------
|   0 | SELECT STATEMENT               |        |
|- * 1 |  HASH JOIN                    |        |
|   2 |   NESTED LOOPS                 |        |
|   3 |    NESTED LOOPS                |        |
|-  4 |     STATISTICS COLLECTOR       |        |
| * 5 |      TABLE ACCESS FULL         | T1     |
| * 6 |     INDEX UNIQUE SCAN          | T2_PK  |
|   7 |    TABLE ACCESS BY INDEX ROWID| T2     |
|-  8 |   TABLE ACCESS FULL            | T2     |
-------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("T1"."ID"="T2"."ID")
   5 - filter("T1"."N"=666)
   6 - access("T1"."ID"="T2"."ID")

Note
```

```
-----
    - this is an adaptive plan (rows marked '-' are inactive)

Adaptive plan:
-------------
This cursor has an adaptive plan, but adaptive plans are enabled for
reporting mode only. The plan that would be executed if adaptive plans
were enabled is displayed below.

Plan hash value: 1837274416

-----------------------------------
| Id  | Operation          | Name |
-----------------------------------
|   0 | SELECT STATEMENT   |      |
|*  1 |  HASH JOIN         |      |
|*  2 |   TABLE ACCESS FULL| T1   |
|   3 |   TABLE ACCESS FULL| T2   |
-----------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
  1 - access("T1"."ID"="T2"."ID")
  2 - filter("T1"."N"=666)

Note
-----
  - this is an adaptive plan
```

为了控制在语句级别是否用了自适应计划，自12.1.0.2起可使用hint (no_)adaptive_plan。

10.4 识别低效的执行计划

遗憾的是，要确定一个执行计划不是最优化的，唯一的办法是找出另一个更好的。虽然如此，简单的检查也可能揭露出暗示低效执行计划的线索。接下来会介绍两种我用于此用途的检查。第13章中介绍了另一种用于评估访问路径效率的检查。

10.4.1 错误的估算

这个检查背后的思路很简单。查询优化器计算成本来决定哪些访问路径、联接顺序以及联接方法应该用于获取一个高效的执行计划。如果成本的计算有误，则很可能查询优化器会选择一个非最优的执行计划。换言之，错误的估算很容易导致选择错误的执行计划。

直接评价一个SQL语句本身的成本在实践中是不可行的。检查查询优化器执行的其他估算则相对容易得多，这种方式的成本估算基于由一个操作返回的行数（基数）。检查估算的基数十分容易，因为你可以使用dbms_xplan包的display_cursor函数，比如说，可以直接使用真实的基数和估算的作对比。就像你刚刚看到的，只有当两个基数值接近时才表明查询优化器工作良好。这种方法的一个核心特性就是不需要SQL语句或数据库结构的相关信息来评价执行计划的优劣。你只需集中精力用实际的数据对比估算的信息。

让我通过一个例子来演示一下这个概念。下面这段来自wrong_estimations.sql脚本输出的摘录展示了一个带有估算（E-Rows）和实际基数（A-Rows）的执行计划。正如你所看到的，操作4的估算是完全错误的（因此还有操作2和操作3）。查询优化器为操作4估算的是，只返回32行数据而不是80 016行。更糟糕的是，操作2和操作3是关联组合操作。这意味着操作6和操作7，实际上被分别执行了80 016和75 808次，而不是估算的只执行32次。这是通过Starts列的值确定的。一定要注意操作6和操作7的估算是正确的。实际上，在作比较之前，实际的基数（A-Rows）必须除以执行的数量（Starts）:

```
SELECT count(t2.col2)
FROM t1 JOIN t2 USING (id)
WHERE t1.col1 = 666

-----------------------------------------------------------------------------
| Id  | Operation                     | Name    | Starts | E-Rows | A-Rows |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |         |      1 |        |      1 |
|   1 |  SORT AGGREGATE               |         |      1 |      1 |      1 |
|   2 |   NESTED LOOPS                |         |      1 |        |  75808 |
|   3 |    NESTED LOOPS               |         |      1 |     32 |  75808 |
|   4 |     TABLE ACCESS BY INDEX ROWID| T1      |      1 |     32 |  80016 |
|*  5 |      INDEX RANGE SCAN         | T1_COL1 |      1 |     32 |  80016 |
|*  6 |     INDEX UNIQUE SCAN         | T2_PK   |  80016 |      1 |  75808 |
|   7 |    TABLE ACCESS BY INDEX ROWID | T2      |  75808 |      1 |  75808 |
-----------------------------------------------------------------------------

  5 - access("T1"."COL1"=666)
  6 - access("T1"."ID"="T2"."ID")
```

要理解这个问题，必须仔细分析为何查询优化器无法计算合理的估算。基数通过将选择率和表中的行数相乘计算得来。因此，如果基数是错误的，引发问题的原因只能有三个：错误的选择率、错误的行数或查询优化器的bug。

在本例中，我们的分析应该始于查看为操作5执行的估算，也就是与"T1"."COL1"=666谓词相关的估算。因为查询优化器基于对象统计信息做估算，那我们来看一下它们是否代表了当前的数据。通过下面的查询，能够获取用于操作5的t1_col1索引的对象统计信息。同时，也可以计算每个键的平均数据行数。这基本上就是在没有直方图可用时查询优化器会使用的值:

```
SQL> SELECT num_rows, distinct_keys, num_rows/distinct_keys AS avg_rows_per_key
  2  FROM user_indexes
  3  WHERE index_name = 'T1_COL1';

NUM_ROWS DISTINCT_KEYS AVG_ROWS_PER_KEY
-------- ------------- ----------------
  160000          5000               32
```

在本例中需要注意的是，平均行数32与上面的执行计划中估算的值一样。要检查这些对象统计信息是否正确，必须将它们与实际数据进行对比。那么，我们在t1表上执行下面的查询。正如你所看到的，该查询不仅计算上一个查询的对象统计信息而且还记录col1列上不等于666的行数:

```
SQL> SELECT count(*) AS num_rows, count(DISTINCT col1) AS distinct_keys,
  2         count(nullif(col1,666)) AS rows_per_key_666
  3  FROM t1;
```

```
NUM_ROWS DISTINCT_KEYS ROWS_PER_KEY_666
-------- ------------- ----------------
  160000          5000            79984
```

从输出中可以确认，对象统计信息是正确的，而且数据倾斜也很严重。因此，直方图对于正确的估算绝对是有必要的。通过下面的查询，可以确认在本例中没有直方图存在：

```
SQL> SELECT histogram, num_buckets
  2  FROM user_tab_col_statistics
  3  WHERE table_name = 'T1' AND column_name = 'COL1';

HISTOGRAM NUM_BUCKETS
--------- -----------
NONE                1
```

收集完缺失的直方图后，查询优化器设法正确地估算基数，进而认为另一个执行计划是最高效的：

```
--------------------------------------------------------------
| Id  | Operation             | Name | Starts | E-Rows | A-Rows |
--------------------------------------------------------------
|   0 | SELECT STATEMENT      |      |      1 |        |      1 |
|   1 |  SORT AGGREGATE       |      |      1 |      1 |      1 |
|*  2 |   HASH JOIN           |      |      1 |  80000 |  75808 |
|*  3 |    TABLE ACCESS FULL| T1   |      1 |  80000 |  80016 |
|   4 |    TABLE ACCESS FULL| T2   |      1 |   151K|   151K|
--------------------------------------------------------------

  2 - access("T1"."ID"="T2"."ID")
  3 - filter("T1"."COL1"=666)
```

注意，在12.1版本不禁用自适应执行计划时执行wrong_estimations.sql脚本时，查询优化器生成了一个自适应执行计划，结果，在运行时自动发现，对于这个查询，散列联接要比嵌套循环更合适。

10.4.2 未识别限制条件

我必须警告你上一节中呈现的检查要优越于本节的。我通常只在执行过第一个检查之后才使用本节的第二个检查。这个检查的思路是验证查询优化器是否正确地识别出SQL语句的限制条件，从而尽可能早地应用它。换言之，检查执行计划是否会导致不必要的处理。

让我通过一个例子来演示一下这个概念，这个例子基于下面的restriction_not_recognized.sql脚本产生输出的摘录。从中，可以看到查询优化器决定以联接t1和t2表开始。第一个联接返回40 000行的结果集。稍后，该结果集与t3表进行联接。只产生了100行的结果集，尽管该操作读取了t3表返回的80 000行数据。这就意味着查询优化器没有识别限制条件，而当大量的处理已经被执行后再应用它就太晚了。顺便说一下，估算联接基数，是查询优化器必须执行的最难的任务之一：

```
SELECT count(t1.pad), count(t2.pad), count(t3.pad)
FROM t1, t2, t3
WHERE t1.id = t2.t1_id AND t2.id = t3.t2_id
```

```
--------------------------------------------------------------------
| Id  | Operation           | Name | Starts | E-Rows | A-Rows |
--------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |      1 |        |      1 |
|   1 |  SORT AGGREGATE     |      |      1 |      1 |      1 |
|*  2 |   HASH JOIN         |      |      1 |  79800 |    100 |
|*  3 |    HASH JOIN        |      |      1 |  40000 |  40000 |
|   4 |     TABLE ACCESS FULL| T1   |      1 |  20000 |  20000 |
|   5 |     TABLE ACCESS FULL| T2   |      1 |  40000 |  40000 |
|   6 |    TABLE ACCESS FULL | T3   |      1 |  80000 |  80000 |
--------------------------------------------------------------------

  2 - access("T2"."ID"="T3"."T2_ID")
  3 - access("T1"."ID"="T2"."T1_ID")
```

遇到这样的问题时，你可能会束手无策。事实上，没有用来描述两张表之间的关系的对象统计信息。修正这种问题的一个可行的办法是使用SQL概要。在本例中应用一个SQL概要会给出如下的执行计划。（我在第11章中介绍了什么是SQL概要以及它是如何工作的。）目前，重要的是要意识到有解决方案存在。要注意，不仅是联接顺序改变了（t2 ➤ t3 ➤ t1），连访问t1表的方式也不同了：

```
------------------------------------------------------------------------------
| Id  | Operation                       | Name  | Starts | E-Rows | A-Rows |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                |       |      1 |        |      1 |
|   1 |  SORT AGGREGATE                 |       |      1 |      1 |      1 |
|   2 |   NESTED LOOPS                  |       |      1 |        |    100 |
|   3 |    NESTED LOOPS                 |       |      1 |    100 |    100 |
|*  4 |     HASH JOIN                   |       |      1 |    100 |    100 |
|   5 |      TABLE ACCESS FULL          | T2    |      1 |  40000 |  40000 |
|   6 |      TABLE ACCESS FULL          | T3    |      1 |  80000 |  80000 |
|*  7 |     INDEX UNIQUE SCAN           | T1_PK |    100 |      1 |    100 |
|   8 |    TABLE ACCESS BY INDEX ROWID| T1    |    100 |      1 |    100 |
------------------------------------------------------------------------------

  4 - access("T2"."ID"="T3"."T2_ID")
  7 - access("T1"."ID"="T2"."T1_ID")
```

10.5 小结

本章描述了如何通过EXPLAIN PLAN语句、动态性能视图、AWR、Statspack以及一些跟踪工具来获取执行计划。正如对前四种技术讨论的那样，对于提取和格式化执行计划，dbms_xplan包是首选的工具。通过它，你能够轻松获取需要的所有信息，从而能够理解执行计划。本章还讨论了一些用于解释执行计划以及用于识别它们是否高效的规则。

很明显，引起性能问题的低效执行计划应该被优化。为此，第四部分开篇通过描述可用的SQL优化技术来介绍这个主题。注意，有多种技术存在着，因为其中每种技术都可以应用于特定的情况或只适用于特定问题的优化。



Part 4 第四部分

优化

工程的目的并不在于获得完美的解决方案，而在于使用有限的资源做到最好。

——兰迪·波许，*The Last Lecture*，2008

只有确定了性能问题的主要原因，你才应该去尝试解决。无论遇到的是什么问题，必须要达到的目的是减少（或者更好的情况——消除）最耗时操作花费的时间。请注意单独一个操作可以由多个动作一个接一个地执行。例如，一个返回多行数据的查询操作会涉及多次获取数据操作。

第 11 章将介绍可用的 SQL 优化技巧，并讲解如何选择它们。第 12 章将介绍解析是如何工作的，如何发现解析的问题，以及如何在不影响性能的情况下减小解析的影响。第 13 章将介绍如何利用可用的访问结构来更有效率地获取单独一张表里的数据。第 14 章将抛开单表，介绍如何多表联合获取数据。第 15 章介绍并行处理和加速流插入的技术，以及减少组件间交互的技术。第 16 章将介绍物理存储参数是如何显著影响性能的。简单地说，这部分章节的主要目的是利用 Oracle 数据库提供的众多特性来缩短操作与 SQL 引擎相互影响的响应时间。

第11章 SQL优化技巧

每当查询优化器无法自动生成有效的执行计划时，就需要手工优化了。表11-1总结了Oracle数据库为此提供的一些技术手段。本章目标不仅是详细介绍这些技巧，而且还会解释每个技巧的作用及其适合的场景。你需要问自己下面三个基础问题来决定使用哪种技巧。

- ❑ SQL语句是否为已知的和静态的？
- ❑ 针对单个会话（或者整个系统），获取到的测量值会影响单条SQL语句还是所有SQL语句？
- ❑ SQL语句可以修改吗？

表11-1 SQL优化技巧及其影响

技　巧	系　统	会　话	SQL 语句	可用版本
修改访问结构	✓			所有版本
修改SQL语句			✓*	所有版本
hint			✓*	所有版本
修改执行环境		✓	✓*	所有版本
存储概要			✓	所有版本
SQL 概要			✓	所有版本††
SQL计划管理			✓	从版本11.1开始‡

* 你必须更改SQL语句才能使用此技巧。

† 需要Tuning Pack，因此需要使用Enterprise Edition。

‡ 需要Enterprise Edition。

让我来解释下这三个问题的重要性。首先，SQL语句有时无法简单获取到，因为它们是在运行时生成的，并几乎在每次执行时都在改变。其他情况下，查询优化器无法正确处理许多SQL语句使用的特殊模式（比如WHERE条件的限制而不能使用索引）。在这些情况下，你需要利用技巧来解决会话或系统级别的问题，而不是SQL语句级别。但这会带来两个问题。一方面，就像表11-1总结的那样，一些技巧只能用在特定的SQL语句上。它们无法在会话或系统级别使用。另一方面，就像第9章解释的那样，当数据库设计良好并且查询优化器正确配置时，通常只需要优化一小部分SQL语句。因此，需要避免技巧影响到由查询优化器自动提供高效执行计划的SQL语句。其次，每当处理不可控的SQL语句应用时（要么是因为代码无法访问，比如包的应用，要么就是SQL语句是在运行时生成的），你都无法使用需要更改代码的技巧。总之，通常你的选择是受限的。

本章的主要目的并非介绍如何找出指定SQL语句的最佳执行计划，例如，介绍特定访问或联

接方法应使用的场景。该分析会在本部分的其他章节介绍。本章的唯一目的是介绍可用的SQL优化技巧。

介绍每一种SQL优化技巧的编排方式都是相同的：先是简介，然后解释该技巧的工作原理，告诉你应该在何时使用它，最后讨论一些常见的误区和谬误。

11.1 修改访问结构

该技巧不是某一特定特性。SQL语句的响应时间不仅非常依赖于存储数据处理的方式，同时也依赖于处理数据的访问方式。

11.1.1 工作原理

怀疑一个SQL语句有性能问题时，首先要做的是确定当前使用的访问结构。基于在数据字典里找到的信息，可以获得以下反馈。

- 涉及的表的组织类型是什么？是堆表、索引组织表还是外部表？或者是存储在群集中的表？
- 物化视图包含的数据是否可用？
- 表、群集和物化视图上存在什么索引？索引都包含了哪些列以及列的排列顺序如何？
- 这些段是如何分区的？

接下来需要评估可用的访问结构是否能够高效处理你要优化的SQL语句。例如，分析期间，你可能会发现对SQL语句的WHERE条件增加索引可以提高效率。假设你在研究以下查询的性能：

```
SELECT *
FROM emp
WHERE empno = 7788
```

基本上，查询优化器会运行下面的执行计划。第一个执行计划执行一次全表扫描，而第二个通过索引访问表。当然，第二个只有在索引存在时才会生成：

```
----------------------------------
| Id  | Operation         | Name |
----------------------------------
|   0 | SELECT STATEMENT  |      |
|   1 |  TABLE ACCESS FULL| EMP  |
----------------------------------

----------------------------------------------
| Id  | Operation                   | Name   |
----------------------------------------------
|   0 | SELECT STATEMENT            |        |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP    |
|   2 |   INDEX UNIQUE SCAN         | EMP_PK |
----------------------------------------------
```

第四部分的多个章节会详细介绍不同的访问结构应该用在何时以及如何使用，因此这里不做过多介绍。现在，重要的是，认识到这是一种基本SQL优化技巧。

11.1.2 何时使用

在适当的位置没有必要的访问结构，或许就不可能优化SQL语句。因此，你需要在任何可以改变访问结构的时候使用该技巧。不幸的是，这并不总是可行，比如当你处理封装的应用并且供应商不支持修改访问结构的时候。

11.1.3 陷阱和谬误

修改访问结构时，必须谨慎处理可能产生的影响。一般来说，每次修改访问结构都会带来正面与负面的影响。实际上，这种影响不太可能只局限于单条SQL语句。只有在少数情况下才不会产生影响。例如，在前面类似例子中要增加索引，就需要考虑索引会减慢索引表上每条INSERT和DELETE语句的执行速度，同样修改索引列的每条UPDATE语句也会产生同样的结果。还应该检查是否有足够的空间来增加访问结构。总的来说，在修改访问结构之前需要仔细判断是否利大于弊。

11.2 修改 SQL 语句

SQL是一种非常强大并且灵活的查询语言。你能够用不同的方法频繁地提交同样的请求。这点对于开发人员来说特别有用。然而对于查询优化器来说，为各种各样的SQL语句提供高效的执行计划才是真正的挑战。请记住，灵活是性能的敌人。

11.2.1 工作原理

举例说，你选择了scott模式下所有没有员工的部门。以下四条SQL语句返回你想要的信息。这些语句都可以在depts_wo_emps.sql脚本中找到：

```
SELECT deptno
FROM dept
WHERE deptno NOT IN (SELECT deptno FROM emp)

SELECT deptno
FROM dept
WHERE NOT EXISTS (SELECT 1 FROM emp WHERE emp.deptno = dept.deptno)

SELECT deptno FROM dept
MINUS
SELECT deptno FROM emp

SELECT dept.deptno
FROM dept, emp
WHERE dept.deptno = emp.deptno(+) AND emp.deptno IS NULL
```

这四条SQL语句的目的是相同的，返回的结果集也是相同的。因此，你或许期望查询优化器为所有的情况提供相同的执行计划。然后这是不可能的，实际上，只有第二条和第四条语句使用相同的执行计划。其他两个完全不同。请注意这些执行计划是在12.1版本中生成的。其他版本会生成不同的执行计划：

```
-----------------------------------------
| Id | Operation          | Name    |
-----------------------------------------
|  0 | SELECT STATEMENT   |         |
|  1 |  HASH JOIN ANTI NA |         |
|  2 |   INDEX FULL SCAN  | DEPT_PK |
|  3 |   TABLE ACCESS FULL| EMP     |
-----------------------------------------
```

```
-----------------------------------------
| Id | Operation          | Name    |
-----------------------------------------
|  0 | SELECT STATEMENT   |         |
|  1 |  HASH JOIN ANTI    |         |
|  2 |   INDEX FULL SCAN  | DEPT_PK |
|  3 |   TABLE ACCESS FULL| EMP     |
-----------------------------------------
```

```
------------------------------------------
| Id | Operation           | Name    |
------------------------------------------
|  0 | SELECT STATEMENT    |         |
|  1 |  MINUS              |         |
|  2 |   SORT UNIQUE NOSORT|         |
|  3 |    INDEX FULL SCAN  | DEPT_PK |
|  4 |   SORT UNIQUE       |         |
|  5 |    TABLE ACCESS FULL| EMP     |
------------------------------------------
```

```
-----------------------------------------
| Id | Operation          | Name    |
-----------------------------------------
|  0 | SELECT STATEMENT   |         |
|  1 |  HASH JOIN ANTI    |         |
|  2 |   INDEX FULL SCAN  | DEPT_PK |
|  3 |   TABLE ACCESS FULL| EMP     |
-----------------------------------------
```

基本上，即使用来访问数据的方法总是相同，用来合并数据产生结果集的方法却不同。在这个特殊案例里，两张表都非常小，因此你不会真正注意到这些执行计划的性能有哪些不同。自然，如果你处理更大的表，就不会是这样了。通常来说，在你处理大量数据时，执行计划里每个细小的不同都可在响应时间和资源利用率上带来本质的不同。

这里的关键点是要明白同样的数据可以由不同的SQL语句提取。要优化一条SQL语句，应该先考虑是否存在其他等价SQL语句。如果存在，请仔细对比它们的执行计划，找出提供最佳性能的那个。

11.2.2 何时使用

只要能够更改SQL语句，都应该考虑使用该技巧。

11.2.3　陷阱和谬误

SQL语句是代码。编写代码的第一条原则就是可维护性。首先，这代表代码应该可读性高并且简明。不幸的是，就像前面解释的那样，最简单或者最可读的SQL编写方法并不总是带来高效的执行计划。因此，某些情况下，为了性能你或许会被迫放弃可读性与简洁性，然而，仅当这样做真正必要且有益时才会这样做。

11.3　hint

根据Merriam-Webster在线字典，hint是一个间接或概要的建议。在Oracle的术语中，hint的定义稍有不同。简单地说，hint是添加到SQL语句中的指令，用来影响查询优化器的判定。换句话说，hint不是仅仅建议某个动作，而是向着该动作推进。在我看来，Oracle选择这个词来命名此功能并不是最佳选择。无论如何，名称并不重要，hint能为你做的才是重要的。不要让名称误导你。

警告　仅因为hint是一个指令，并不代表查询优化器就总是会使用它。或者反过来说，仅因为查询优化器不使用hint，并不代表hint仅仅是一个建议。就像我稍后将介绍的，有些案例里，hint只是不相关或不合法，因此不会影响查询优化器生成的执行计划。

11.3.1　工作原理

接下来的部分介绍hint是什么，hint的分类及如何使用它们。在讨论细节之前需要注意，使用hint要比你想象得重要。实际上，在实践中，hint被错误使用是很常见的。

1. 什么是hint

当处理一条SQL语句时，查询优化器会考虑许多种执行计划。理论上，它会考虑所有可行的执行计划。实际上，除了简单的SQL语句之外，优化器为了保持合理的优化时间，不会考虑太多种组合。因此，查询优化器会根据推断排除某些执行计划。当然，完全忽略一些执行计划的决定很关键，并且这么做查询优化器的可信性也会受到怀疑。

指定一个hint时，你的目的要么是改变执行环境，启用或者禁用某个特性，要么是降低查询优化器需要考虑的执行计划数量。除非改变执行环境，使用hint你将告诉查询优化器，针对某条特定SQL语句应该考虑哪些操作或不应该考虑哪些操作。例如，查询优化器要为以下查询生成执行计划：

```
SELECT *
FROM emp
WHERE empno = 7788
```

如果emp表是堆表并且empno列有索引，那么查询优化器至少考虑两种执行计划。第一种通过全表扫描彻底读了一遍emp表：

```
----------------------------------
| Id  | Operation          | Name |
----------------------------------
```

```
|   0 | SELECT STATEMENT  |       |
|   1 |  TABLE ACCESS FULL| EMP   |
----------------------------------
```

第二种是基于WHERE子句（empno=7788）的谓词做一次索引查找，然后通过在索引里找到的rowid去访问表中的数据：

```
---------------------------------------------
| Id  | Operation                   | Name   |
---------------------------------------------
|   0 | SELECT STATEMENT            |        |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP    |
|   2 |   INDEX UNIQUE SCAN         | EMP_PK |
---------------------------------------------
```

在这样的案例里，要控制查询优化器提供的执行计划，你可以加入hint来指定使用全表扫描或者索引扫描。重要的是需要明白你不能告诉查询优化器，“我想要在emo表上执行全表扫描，所以去搜一个包含它的执行计划”。然而，你可以告诉它，“如果需要在对emp表执行全表扫描还是索引扫描之间做出选择，请选择全表扫描”。这是一个轻量的但本质上的不同。当查询优化器必须在几个可能的执行计划间选择时，hint可以允许你影响它的选择。

为了进一步强调这点，让我们来看一个基于图11-1显示的决策树的例子。请注意，即使查询优化器利用决策树，这也只是个一般的例子，并没有与Oracle数据库有直接关系。在图11-1中，目的是从决策树的根节点(1)向下终止于叶子节点（111~123）。换句话说，目的是从点A到点B选择一条路径。由于某些原因，这必定会经过节点122的。要这么做，在Oracle的语法里就需要两个hint，加入来修剪从节点12到节点121和节点123的路径。从节点12到节点122只会存在这唯一的一条路径。但这并不足以保证路径经过节点122。实际上，如果节点1经过节点11而不是节点12，那么这两个hint就不会起作用。因此，要引导路径通过节点122，你需要增加另外一个hint来修剪从节点1到节点11的路径。

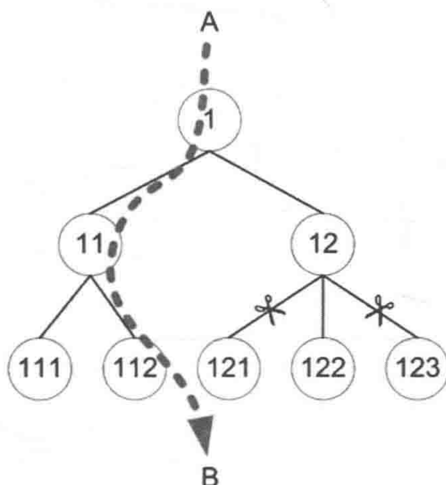


图11-1　修剪决策树

查询优化器也会发生类似的情况。实际上，只有在查询优化器决定了应用hint的选择后才会对它

做评估。因此，一旦指定了一个hint，你或许会被迫加入几个hint来确保它正常工作。并且在实践中，随着执行计划复杂度的增加，想要找到所有可用的hint来获得想要的执行计划会变得越来越困难。

2. 指定hint

hint是Oracle的扩展。为了不影响SQL语句与其他数据库引擎的兼容性，Oracle决定把它们作为一种特殊的注释来加入。注释与hint仅有的不同如下所示。

- hint必须紧随DELETE、INSERT、MERGE、SELECT和UPDATE关键字。换句话说，它们不能像注释那样指定在SQL语句的任意位置。
- 注释分隔符的第一个字符必须是加号（+）。

一般而言，hint的语法错误不会引发报错。如果解析器无法解析它们，就会把它们当作注释。有时，注释与hint混合同样可行。下面的两个例子展示了如何使用上一节介绍的查询强制在emp表上执行全表扫描：

```
SELECT /*+ full(emp) */ *
FROM emp
WHERE empno = 7788

SELECT /*+ full(emp) you can add a real comment after the hint */ *
FROM emp
WHERE empno = 7788
```

然而，混合注释与hint并不总是可行的。例如，注释加在hint前面就会使hint失效。以下查询展示了这样的案例：

```
SELECT /*+ but this one does not work full(emp) */ *
FROM emp
WHERE empno = 7788
```

因为注释能使hint失效，所以不建议将注释与hint混合使用。最好是分开它们。

3.hint的类别

划分hint的类别有好几种方法（观点）。个人而言，我喜欢按以下类别对它们进行分组。

- *初始化参数*hint（initialization parameter hint）会重写一些在系统或会话级别定义的初始化参数的设置。我将以下hint划分在这个类别里：all_rows、cursor_sharing_exact、dynamic_sampling、first_rows、gather_plan_statistics、optimizer_features_enable和opt_param。我会在11.4节中介绍这些hint，并且我已在第10章中介绍了gather_plan_statistics hint。请注意，当指定这些hint时，它们总是会重写实例或会话级别的值。
- *查询转换*hint（query transformation hint）控制着逻辑优化期间查询转换技术的利用率。我将以下hint划分在这个类别里：(no_)eliminate_join、no_expand、(no_)expand_table、(no_)fact、(no_)merge、(no_)outer_join_to_inner、(no_)rewrite、(no_)star_transformation、(no_)unnest、no_xmlindex_rewrite、no_xml_query_rewrite和use_concat。我会在后续几节介绍部分hint，其他hint会在第14章和第15章中进行介绍。
- *访问路径*hint（access path hint）控制着用来访问数据的方法（例如，是否使用索引）。我将以下hint划分在这个类别里：cluster、full、hash、(no_) index、index_asc、index_combine、

index_desc、(no_) index_ffs、index_join、(no_) index_ss、index_ss_asc和index_ss_desc。我会在第13章介绍这些hint及其访问方法。

- 联接hint(join hint)不仅控制着联接方法，也包含用来联接表的顺序。我将以下hint划分在这个类别里：leading、(no_)nlj_batching、ordered、(no_)swap_join_inputs、(no_)use_cube、(no_)use_hash、(no_)use_merge、use_merge_cartesian、(no_)use_nl和use_nl_with_index。我会在第14章介绍这些hint及其联接方法。
- 并行处理hint(parallel processing hint)控制如何使用以及是否使用并行处理。我将以下hint划分在这个类别里：(no_) parallel、(no_) parallel_index、(no_) pq_concurrent_union、pq_distribute、pq_filter、(no_)pq_skew、(no_)px_join_filter和(no_)statement_queuing。我会在第15章介绍这些hint及其并行处理。在第14章中会随同智能分区联接一起提供pq_distribute hint的一个可能利用率。
- 其他hint控制着不属于上面任何类别的其他特性。我将以下hint划分在这个类别里：(no) append、append_values、(no_)bind_aware、(no_)result_cache、(no)cache、change_dupkey_error_index、driving_site、(no_) gather_optimizer_statistics、ignore_row_on_dupkey_index、inline、materialized、(no_)monitor、model_min_analysis、(no_)monitor、qb_name和retry_on_row_change。我会在本章稍后介绍qb_name hint，其他的一些hint的介绍会贯穿在整本书中。

尽管通过本书我介绍或展示了很多hint的例子，但我并未提供真实的参考或它们完整的语法。此类参考在*Oracle Database SQL Lanaguage Reference*手册的第2章中提供。

值得指出的是，存在大量hint会禁用某个特殊操作或特性(no_前缀的hint)。好处是有时指定某些操作或特性不可使用要更容易。

以上提供的hint列表并不完整，它们只介绍了记录在*SQL Reference Guide*手册中的部分。还有很多hint并未记录在文档里。你会在稍后的11.6节中看到部分hint。从11.1版本起，可以查询v$sql_hint视图来获取接近完整的hint列表。

4.hint的有效性

简单的SQL语句只有单个查询块。当使用视图或集合时才会存在多个查询块，如子查询、内敛视图和集合运算。例如，以下查询有两个查询块(仅仅出于演示的目的，我使用子查询来替代一个真实的视图)。第一个查询块是引用dept表的主查询。第二个是引用emp表的子查询：

```
WITH
  emps AS (SELECT deptno, count(*) AS cnt
           FROM emp
           GROUP BY deptno)
SELECT dept.dname, emps.cnt
FROM dept, emps
WHERE dept.deptno = emps.deptno
```

通常情况下，初始化参数hint对整个SQL语句都有效(dynamic_sampling是个例外)。其他大多数hint仅对单个查询块有效(有两个例外，bind_aware和monitor)。对单个查询块有效的hint必须指定在它们控制的块内。例如，如果想让上个查询里的两张表都指定访问路径hint，那么一个hint需要加在主查询里，另一个需要加在子查询里。它们的有效性仅限于它们定义的查询块中：

```
WITH
  emps AS (SELECT /*+ full(emp) */ deptno, count(*) AS cnt
           FROM emp
           GROUP BY deptno)
SELECT /*+ full(dept) */ dept.dname, emps.cnt
FROM dept, emps
WHERE dept.deptno = emps.deptno
```

这条规则的例外是全局hint（global hint）。使用全局hint时，有可能通过使用点记法引用包含在其他查询块中的对象（如果已命名它们）。例如，下面的SQL语句，主查询包含作用于子查询的hint。请注意子查询对引用名称的使用：

```
WITH
  emps AS (SELECT deptno, count(*) AS cnt
           FROM emp
           GROUP BY deptno)
SELECT /*+ full(dept) full(emps.emp) */ dept.dname, emps.cnt
FROM dept, emps
WHERE dept.deptno = emps.deptno
```

全局hint的语法支持超过两层级别的引用（例如，一个视图引用自另一个视图）。对象必须要用点分隔开（例如，view1.view2.view3.table）。

提示　全局hint并非总处理某些查询转换，我建议你使用基于查询块名称（立刻显示）的语法。

由于WHERE子句的子查询不能命名，因此它们的对象无法被全局hint引用。为了解决这个问题，有另一种方法可以达到此目的。实际上，大多数hint可以接受一个参数，这个参数指定这些hint对哪个查询块有效。这样的话，hint可以在SQL语句开头被分组并且仅引用它们应用的查询块。要使用这些引用，不仅需要查询优化器对每个查询块生成一个查询块名称（query block name），而且允许你使用qb_name hint来自定义名称。例如，下面的查询，两个查询块分别叫main和sq。接着在full hint里，查询块名称通过前缀@标识来引用。请注意在主查询中指定对emp表进行子查询的访问路径hint：

```
WITH
  emps AS (SELECT /*+ qb_name(sq) */ deptno, count(*) AS cnt
           FROM emp
           GROUP BY deptno)
SELECT /*+ qb_name(main) full(@main dept) full(@sq emp) */ dept.dname, emps.cnt
FROM dept, emps
WHERE dept.deptno = emps.deptno
```

上一个例子显示了如何指定自己的名称。现在让我们来看看如何使用查询优化器生成的名称。首先，你必须知道它们是什么。为此，你可以使用EXPLAIN PLAN语句和dbms_xplan包，如下面的例子所示。请注意，alias选项被传递给display函数以确保查询块名称和别名是输出的一部分：

```
SQL> EXPLAIN PLAN FOR
  2  WITH emps AS (SELECT deptno, count(*) AS cnt
  3                FROM emp
  4                GROUP BY deptno)
  5  SELECT dept.dname, emps.cnt
  6  FROM dept, emps
```

```
  7 WHERE dept.deptno = emps.deptno;

SQL> SELECT * FROM table(dbms_xplan.display(NULL, NULL, 'basic +alias'));
-------------------------------------
| Id  | Operation             | Name |
-------------------------------------
|   0 | SELECT STATEMENT      |      |
|   1 |  HASH JOIN            |      |
|   2 |   VIEW                |      |
|   3 |    HASH GROUP BY      |      |
|   4 |     TABLE ACCESS FULL| EMP  |
|   5 |   TABLE ACCESS FULL  | DEPT |
-------------------------------------

Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------

  1 - SEL$2
  2 - SEL$1 / EMPS@SEL$2
  3 - SEL$1
  4 - SEL$1 / EMP@SEL$1
  5 - SEL$2 / DEPT@SEL$2
```

系统生成的查询块名称由前缀和字符串组成。前缀是根据查询块里的操作生成的。表11-2做了总结。字符串是查询块的编号，基于它们解析SQL语句时所在的位置（左或右）。在前面的例子中，主查询块被命名为SEL$2，子查询块被命名为SEL$1。

表11-2　前缀在查询块名称中的使用

前　缀	用　途
CRI$	CREATE INDEX语句
DEL$	DELETE语句
INS$	INSERT语句
MISC$	其他SQL语句，比如LOCK TABLE
MRC$	MERGE语句
SEL$	SELECT语句
SET$	集合运算符，比如：UNION和MINUS
UPD$	UPDATE语句

如下所示，系统生成的查询块名称的利用率与用户定义的查询块名称的利用率并无不同：

```
WITH
  emps AS (SELECT deptno, count(*) AS cnt
           FROM emp
           GROUP BY deptno)
SELECT /*+ full(@sel$2 dept) full(@sel$1 emp) */ dept.dname, emps.cnt
FROM dept, emps
WHERE dept.deptno = emps.deptno
```

我需要对查询转换期间生成的查询块名称做最后一次解释。由于它们不是SQL语句的一部分，因而它们无法像其他对象那样计数。因此，查询优化器会为它们生成8位的散列值。下面的例子展示了

这种情况。这里系统生成的查询块名称为SEL$5DA710D3：

```
SQL> EXPLAIN PLAN FOR
  2  SELECT deptno
  3  FROM dept
  4  WHERE NOT EXISTS (SELECT 1 FROM emp WHERE emp.deptno = dept.deptno);

SQL> SELECT * FROM table(dbms_xplan.display(NULL,NULL,'basic +alias'));

-----------------------------------
| Id  | Operation           | Name |
-----------------------------------
|   0 | SELECT STATEMENT    |      |
|   1 |  HASH JOIN ANTI     |      |
|   2 |   TABLE ACCESS FULL| DEPT |
|   3 |   TABLE ACCESS FULL| EMP  |
-----------------------------------

Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------

   1 - SEL$5DA710D3
   2 - SEL$5DA710D3 / DEPT@SEL$1
   3 - SEL$5DA710D3 / EMP@SEL$2
```

在前面的输出中，会发现一件有趣的事情，当查询转换发生时，执行计划里的一些行（比如第二行）会有两个查询块名称。它们都可以使用hint。但是从查询优化器的角度来看，仅当完全相同的查询转换发生时，查询转换之后的查询块名称（这里是SEL$5DA710D3）才可用。

11.3.2　何时使用

hint的目的有两个。首先，当查询优化器不能自动生成有效的执行计划时，它们就成了方便的变通方法。这种情况下，你将用它们得到一个更好的执行计划。这里要强调的是，hint是一种变通方法，因此不应该用在长期的解决方案里。然而，在某些情况下，它们是解决问题的唯一可行方法。其次，在查询优化器对生成的执行计划二选一时，hint对评估选择很有帮助。这种情况下，可以使用它们来做模拟分析。

11.3.3　陷阱和谬误

每当你想通过访问路径hint、联接hint或并行处理hint来锁定某个特定的执行计划时，必须指定足够的hint来实现稳定性。这里的稳定性表示即使在一定程度上，对象的统计信息和访问结构发生了改变，执行计划也不会改变。要锁定某一执行计划，正常情况下不仅要给SQL语句里的每张表添加访问路径hint，还要添加多个联接hint来控制联接方法和顺序。请注意，其他类型的hint（比如，初始化参数hint和查询转换hint）通常不会受到此问题的影响。

当处理SQL语句时，解析器会检查hint的语法。尽管如此，当发现hint有无效的语法时，除非是行为奇怪的change_dupkey_error_index、ignore_row_on_dupkey_inddex和retry_on_row_change等hint，否则并不会引发报错。这代表解析器把这个假hint当作注释来处理。从一方面看，仅仅因为输入错误造

成hint不可使用很让人恼火。但另一方面，这对已经部署好的应用有益，比如经常会在hint里引用对象（比如，index hint会引用索引名称）或升级到新的数据库版本，都不会因为访问结构的改变而引起中断。即便如此，我还是想要一种可以检验SQL语句里hint的方法。比如，通过EXPLAIN PLAN语句，在这方面提供一个警告（比如，在dbms_xplan的输出中多了条记录）是非常简单的。我所知道的唯一能部分实现该功能的方法就是设置10132事件。实际上，该事件生成的数据结尾部分就是留给hint的。你可以在这部分检查两件事。首先，每个hint都会被列出。如果有hint不在，这代表并没有被识别。其次，检查是否存在一条通知某些hint有错误的消息。（在这种情况下，会将err字段设置为大于0的值）。请注意，为了获得下面的输出，已指定了两个彼此冲突的初始化参数：

```
Dumping Hints
=============
  atom_hint=(@=0x6b796498 err=4 resol=0 used=0 token=454 org=1 lvl=1 txt=ALL_ROWS )
  atom_hint=(@=0x6b796578 err=4 resol=0 used=0 token=453 org=1 lvl=1 txt=FIRST_ROWS )
********** WARNING: SOME HINTS HAVE ERRORS *********
```

请注意，使用这种方法的话，hint语法正确，但是引用了错误的对象并不会报错。因此，这不是最终解决方案。

hint使用中最常见的错误都与表别名有关。规则是当hint引用一张表时，只要表有别名就应该使用别名代替表名。在下面的例子中，可以看到如何为emp表定义别名（e）。在这种情况下，当full hint引用表名时，这个hint不会起作用。请注意，在第一个例子里使用了索引扫描而不是期望的全表扫描：

```
SQL> EXPLAIN PLAN FOR SELECT /*+ full(emp) */ * FROM emp e WHERE empno = 7788;

SQL> SELECT * FROM table(dbms_xplan.display(NULL,NULL,'basic'));

---------------------------------------------
| Id  | Operation                   | Name   |
---------------------------------------------
|   0 | SELECT STATEMENT            |        |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP    |
|   2 |   INDEX UNIQUE SCAN         | EMP_PK |
---------------------------------------------

SQL> EXPLAIN PLAN FOR SELECT /*+ full(e) */ * FROM emp e WHERE empno = 7788;

SQL> SELECT * FROM table(dbms_xplan.display(null,null,'basic'));

----------------------------------
| Id  | Operation         | Name |
----------------------------------
|   0 | SELECT STATEMENT  |      |
|   1 |  TABLE ACCESS FULL| EMP  |
----------------------------------
```

升级期间应该检查hint的影响，但是却总是会被忘记。查询优化器不会自动提供高效的执行计划，而是根据查询优化器使用的决策树类型来决定效果，这时hint就是一种方便的变通方法；任何时候加过hint的SQL语句要在另一个版本的数据库（并且因此使用了另一版本的查询优化器）上执行，都要进行仔细检查。换句话说，当检验新版本数据库上的应用时，最好的解决方案是重新检查并且重测所有包含hint的SQL语句。出于测试目的，也应该在会话级别把未公开的初始化参数_optimizer_

ignore_hints设置为TRUE来禁用所有的hint。请注意，要避免在系统级别设置该参数，因为数据库引擎自身也会用到许多hint。

由于视图可能会用于不同的环境，因此不建议在视图中指定hint。如果真的想在视图中添加hint，确保加入的hint对所有模块都有意义。

11.4 修改执行环境

第9章介绍了如何配置查询优化器。该配置就是所有用户连接到数据库引擎的默认执行环境。因此，它必须适合于大多数情况。当多个应用使用数据库时（例如，由于数据库服务器整合）或单个应用基于模块需要不同环境时（例如，白天的OLTP和夜晚的批处理），单独一个环境无法满足所有场景是很常见的。这种情况下，在会话级别甚至SQL语句级别修改执行环境是恰当的。

11.4.1 工作原理

在会话级别修改执行环境与在SQL语句级别修改完全不同。因此，我会分别介绍两种情况。此外，我会介绍一些显示数据库实例、单个会话或子游标相关环境的动态性能视图。

1. 会话级别

第9章介绍的大多数初始化参数都可以在会话级别使用ALTER SESSION语句进行修改。因此，如果你有用户或者模块需要特殊配置，可以简单地在会话级别更改默认值。例如，根据连接到数据库的用户来设置执行环境，可以使用配置表和数据库触发器，如下面的例子所示。可以在exec_env_trigger.sql脚本中找到该SQL语句：

```
CREATE TABLE exec_env_conf (username  VARCHAR2(30),
                            parameter VARCHAR2(80),
                            value     VARCHAR2(512))

CREATE OR REPLACE TRIGGER execution_environment AFTER LOGON ON DATABASE
BEGIN
  FOR c IN (SELECT parameter, value
            FROM exec_env_conf
            WHERE username = sys_context('userenv','session_user'))
  LOOP
    EXECUTE IMMEDIATE 'ALTER SESSION SET ' || c.parameter || '=' || c.value;
  END LOOP;
END;
```

接着针对需要某个特别配置的每个用户，应该为每个初始化参数在配置表里插入一行数据。例如，当名叫Alberto的用户登录时，下面两个INSERT语句会在会话级别更改和定义两个参数：

```
INSERT INTO exec_env_conf VALUES ('ALBERTO', 'optimizer_mode', 'first_rows_10')

INSERT INTO exec_env_conf VALUES ('ALBERTO', 'optimizer_dynamic_sampling', '0')
```

当然，也可以为单个模式定义触发器，或执行基于诸如userenv上下文的其他检查。

2. SQL语句级别

SQL语句级别的执行环境通过初始化参数hint来更改。由于使用hint，因此之前介绍的hint的行为和性能都会生效。

并不是所有的初始化参数组成的查询优化器配置都可以在SQL语句级别上修改。表11-3总结了在SQL语句级别上哪些参数和值与初始化参数hint一样可以实现相同的配置。请注意对于某些初始化参数（比如，cursor_sharing）来说，并不是所有的值都可以使用hint来设置。

表11-3 SQL语句级别hint可修改的查询优化器配置

初始化参数	hint
cursor_sharing=exact	cursor_sharing_exact
optimizer_dynamic_sampling=x	dynamic_sampling(x)
optimizer_features_enable=x	optimizer_features_enable('x')
optimizer_features_enable not set	optimizer_features_enable(default)
optimizer_index_caching=x	opt_param('optimizer_index_caching' x)
optimizer_index_cost_adj=x	opt_param('optimizer_index_cost_adj' x)
optimizer_mode=all_rows	all_rows
optimizer_mode=first_rows	first_rows
optimizer_mode=first_rows_x	first_rows(x)
optimizer_secure_view_merging=x	opt_param('optimizer_secure_view_merging' 'x')
optimizer_use_pending_statistics=x	opt_param('optimizer_use_pending_statistics' 'x')
result_cache_mode=manual	no_result_cache
result_cache_mode=force	result_cache
star_transformation_enabled=x	opt_param('star_transformation_enabled' 'x')

3. 动态性能视图

有以下三个动态性能视图提供执行环境的信息。

- v$sys_optimizer_env提供实例级别的执行环境信息。例如，可以找出哪个初始化参数没有设置成默认值：

```
SQL> SELECT name, value, default_value
  2  FROM v$sys_optimizer_env
  3  WHERE isdefault = 'NO';

NAME                        VALUE DEFAULT_VALUE
--------------------------- ----- -------------
star_transformation_enabled true  false
```

- v$ses_optimizer_env提供每个会话的执行环境信息。由于没有列提供某个初始化参数是否在系统或会话级别被修改的信息，因此可以使用以下查询达到目的：

```
SQL> SELECT name, value
  2  FROM v$ses_optimizer_env
  3  WHERE sid = 124 AND isdefault = 'NO'
  4  MINUS
  5  SELECT name, value
  6  FROM v$sys_optimizer_env;

NAME             VALUE
```

11

```
-------------- ------------
cursor_sharing force
optimizer_mode first_rows_10
```

- v$sql_optimizer_env提供库缓存中存在的每个子游标的执行环境信息。比如，以下查询可以查明同一父游标的两个子游标是否使用不同的执行环境：

```
SQL> SELECT e0.name, e0.value AS value_child_0, e1.value AS value_child_1
  2  FROM v$sql_optimizer_env e0, v$sql_optimizer_env e1
  3  WHERE e0.sql_id = e1.sql_id
  4  AND e0.sql_id = 'a5ks9fhw2v9s1'
  5  AND e0.child_number = 0
  6  AND e1.child_number = 1
  7  AND e0.name = e1.name
  8  AND e0.value <> e1.value;

NAME                 VALUE_CHILD_0 VALUE_CHILD_1
-------------------- ------------- -------------
hash_area_size       33554432      131072
optimizer_mode       first_rows_10 all_rows
cursor_sharing       force         exact
workarea_size_policy manual        auto
```

11.4.2　何时使用

每当默认配置无法满足应用的某一部分或部分用户时，就应该修改默认配置。尽管在会话级别初始化参数随时都可以修改，但hint只有在修改SQL语句级别时才有效。

11.4.3　陷阱和谬误

可以将设置集中在数据库或应用中时，在会话级别修改执行环境是非常简单的。如果使用的应用或模块共享的连接池需要不同的执行环境，你需要额外注意。实际上，会话参数与物理连接有关。由于其他应用或模块会使用物理连接，每次从连接池获取到连接都要设置一次执行环境（当然代价很高，因为需要额外反复连接数据库）。如果有的应用或者模块需要不同的执行环境，为了避免这种开销，应该使用不同的连接池和不同的用户。这样，就可以针对每个连接池使用单独的配置，并且通过定义不同的用户连接到数据库，你也许能够将配置集中到一个简单的数据库触发器中。

在SQL语句级别修改执行环境也存在与hint一样的误区和谬误。

11.5　存储概要

存储概要的作用是，在执行环境或对象统计信息中存在更改时，提供稳定的执行计划。为此，这个功能也称为计划稳定性（plan stability）。在Oracle文档中记录了体现该功能优势的两个重要场景。第一个是从基于规则的优化器（RBO）向基于成本的优化器（CBO）的迁移。第二个场景是将Oracle数据库升级到新版本。在这两个场景中，目的都是在应用使用旧配置或版本时存储关于执行计划的信息，然后使用该信息来提供与新的配置或版本相同的执行计划。不幸的是，实际上即使正确地使用存储概

要（stored outline），你仍能看到执行计划在改变。或许是由于这个原因，我从未见过哪个数据库大范围地使用存储概要。因此，实际上存储概要仅会用在某些具体的SQL语句上。

注意 从11.1版本之后，存储概要不再支持**SQL 计划管理**（SQL plan management）（本章稍后会介绍）。

11.5.1 工作原理

接下来的几部分会介绍什么是存储概要以及如何使用它们。

1. 什么是存储概要

存储概要是与SQL语句相关联的对象，其作用是在为SQL语句生成执行计划时影响查询优化器。更具体地说，存储概要是一组hint，或者更准确地说，是所有能强制查询优化器始终为给定SQL语句生成特定执行计划的hint组合。

注意 并不是所有hint都可以保存在存储概要中。要想知道不能保存哪些hint，可以执行以下查询：

```
SELECT name FROM v$sql_hint WHERE version_outline IS NULL
```

尽管大多数无法保存到存储概要中的hint不会影响执行计划（例如gather_plan_statistics），但有些还是会的（例如materialize和inline）。因此，有些执行计划因无法在SQL语句中指定hint而不能通过存储概要固定。

存储概要的优势之一是，当它应用于某个SQL语句时，你并不需要为了应用存储概要而修改SQL语句。存储概要保存在数据字典里，并且查询优化器会自动选择它们。图11-2显示了在选择期间执行的基本步骤。首先，会将SQL语句中的空格移除，进行标准化，并将非文字字符串转换为大写。作为结果的SQL语句签名（SQL语句文本的散列值）会被计算。接着，根据签名，在数据字典里执行查找。每当找到包含同样签名的存储概要时，就会执行检查来确保这个SQL语句是最优的，并且与绑定存储概要的SQL语句是等价的。这一步很重要，因为签名是散列值，可能会产生冲突。如果测试成功，那么hint组成的存储概要就会包含在生成的执行计划里。

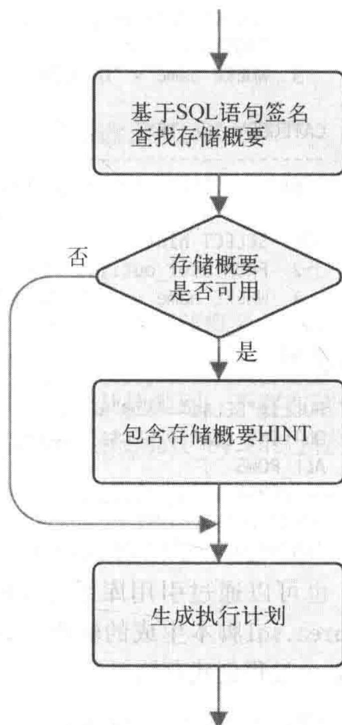


图11-2 选择存储概要期间要执行的主要步骤

2. 创建存储概要

可以使用两种方法来创建存储概要。数据库自动创建和手工创建。如果想为指定会话甚至整个系统执行的每条SQL语句创建存储概要，可以使用第一种方法。然而，就像前面提到的，通常没有必要

这么做。因此，经常会手工创建它们。

要激活自动创建，需要将初始化参数create_stored_outlines设置为TRUE或者指定一个类别（category）。使用类别的目的是要集合多个存储概要来实现统一管理。将初始化参数设置为TRUE时会使用默认类别，其名称为DEFAULT。可以在会话级别和系统级别动态更改该初始化参数。要禁用自动创建，需要将初始化参数设置为FALSE。

要手工创建存储概要，必须使用CREATE OUTLINE语句。下面的SQL语句，摘录自outline_from_text.sql脚本，展示了名为outline_from_text的存储概要的创建，该存储概要与test类别相关联，并基于ON子句中指定的查询：

```
CREATE OR REPLACE OUTLINE outline_from_text
FOR CATEGORY test
ON SELECT * FROM t WHERE n = 1970
```

一旦创建好，就可以通过user_outlines和user_outline_hints视图来显示存储概要的信息和它们的属性（对于这两个视图，也存在以all、dba开头的视图，同时，在12.1多租户环境下还有以cdb开头的视图）。User_outlines视图显示除了hint以外的信息。下面的查询显示的信息为上一个SQL语句创建的存储概要：

```
SQL> SELECT category, sql_text, signature
  2  FROM user_outlines
  3  WHERE name = 'OUTLINE_FROM_TEXT';

CATEGORY SQL_TEXT                       SIGNATURE
-------- ------------------------------ --------------------------------
TEST     SELECT * FROM t WHERE n = 1970 73DC40455AF10A40D84EF59A2F8CBFFE

SQL> SELECT hint
  2  FROM user_outline_hints
  3  WHERE name = 'OUTLINE_FROM_TEXT';

HINT
------------------------------------
FULL(@"SEL$1" "T"@"SEL$1")
OUTLINE_LEAF(@"SEL$1")
ALL_ROWS
DB_VERSION('11.2.0.3')
OPTIMIZER_FEATURES_ENABLE('11.2.0.3')
IGNORE_OPTIM_EMBEDDED_HINTS
```

也可以通过引用库缓存里的游标来手工创建存储概要。下面的例子，摘录自outline_from_sqlarea.sql脚本生成的输出，显示了如何从库缓存里选择游标并且通过dbms_outln包下的create_outline过程创建存储概要：

```
SQL> SELECT hash_value, child_number
  2  FROM v$sql
  3  WHERE sql_text = 'SELECT * FROM t WHERE n = 1970';

HASH_VALUE CHILD_NUMBER
---------- ------------
 308120306            0
```

```
SQL> BEGIN
  2    dbms_outln.create_outline(hash_value   => '308120306',
  3                              child_number => 0,
  4                              category     => 'test');
  5  END;
  6  /
```

警告 create_outline过程不会基于与引用的游标相关联的执行计划创建存储概要。相反，它接受与游标相关联的SQL语句的文本并重解析它。因此，与存储概要相关联的执行计划并不需要和与游标相关联的执行计划一致。例如，一个不同的执行环境可以很容易导致另一个执行计划。

如下所示，create_outline过程仅接受三个参数。这代表存储概要的名称是自动生成的。要找出系统生成的名称，需要查询视图，比如user_outlines。下面的查询返回最后创建的存储概要名：

```
SQL> SELECT name
  2  FROM user_outlines
  3  WHERE timestamp = (SELECT max(timestamp) FROM user_outlines);

NAME
------------------------------
SYS_OUTLINE_13072411155434901
```

系统自动生成的存储概要名称是可以自定义的。下一部分将介绍如何修改。

3. 修改存储概要

要更改存储概要名，需要执行ALTER OUTLINE语句：

```
ALTER OUTLINE SYS_OUTLINE_13072411155434901 RENAME TO outline_from_sqlarea
```

使用ALTER OUTLINE语句或dbms_outln包下的update_by_cat过程，也可以修改存储概要的类别。然而前者修改单个存储概要的类别，后者把所有属于一个类别的存储概要都移动到另一个类别中。可是由于bug 5759631，使用ALTER OUTLINE不能修改存储概要类别DEFAULT（对于其他类别，不存在这个问题）。下面的例子介绍了当你尝试修改时会发生什么，同时还介绍了如何使用update_by_cat过程执行同样的操作：

```
SQL> ALTER OUTLINE outline_from_text CHANGE CATEGORY TO DEFAULT;
ALTER OUTLINE outline_from_text CHANGE CATEGORY TO DEFAULT
                                                   *
ERROR at line 1:
ORA-00931: missing identifier

SQL> execute dbms_outln.update_by_cat(oldcat => 'TEST', newcat => 'DEFAULT')

SQL> SELECT category
  2  FROM user_outlines
  3  WHERE name = 'OUTLINE_FROM_TEXT';

CATEGORY
--------
```

```
DEFAULT
```

最后，使用ALTER OUTLINE语句，也可以生成存储概要，就像重建一样。通常情况下，会在想要查询优化器生成一组新的hint时使用该语句。如果更改了与存储概要相关的对象的访问结构，可能有必要使用该语句：

```
ALTER OUTLINE outline_from_text REBUILD
```

4. 激活存储概要

只有在存储概要被激活后查询优化器才会处理。要激活它，存储概要需要满足两个条件。第一，存储概要必须是启用的。在创建存储概要时，默认是启用的。要启用和停用存储概要，可以使用ALTER OUTLINE语句：

```
ALTER OUTLINE outline_from_text DISABLE

ALTER OUTLINE outline_from_text ENABLE
```

第二个条件是类别（category）必须在会话或系统级别通过初始化参数use_stored_outlines来激活。初始化参数可以接受的值为TRUE、FALSE或类别名。如果指定TRUE，类别默认值为DEFAULT。以下SQL语句在会话级别激活属于test类别的存储概要：

```
ALTER SESSION SET use_stored_outlines = test
```

由于初始化参数use_stored_outlines只支持单个类别,因此在同一时间一个会话只能激活一个类别。

要想知道查询优化器是否使用了存储概要，可以利用dbms_xplan包下的函数。实际上，正如下面的例子所示，输出的Note部分明确提供了需要的信息：

```
SQL> EXPLAIN PLAN FOR SELECT * FROM t WHERE n = 1970;

SQL> SELECT * FROM table(dbms_xplan.display);

---------------------------------
| Id  | Operation         | Name |
---------------------------------
|   0 | SELECT STATEMENT  |      |
|*  1 |  TABLE ACCESS FULL| T    |
---------------------------------

  1 - filter("N"=1970)

Note
-----
   - outline "OUTLINE_FROM_TEXT" used for this statement
```

对于库缓存中存储的游标，v$sql视图的outline_category列会指明在执行计划生成期间是否使用了存储概要。不幸的是，这只给出了类别名。存储概要名本身却是未知的。如果没有使用存储概要，该列将会是NULL。

有一种方法可以知道在一段时间内是否使用过存储概要，可以使用dbms_outln包下的clear_used过程来重置使用标记。接着，稍后再查看该标记，就可以判断是否使用了这个存储概要。然而，并不会给出更多的使用信息（比如，使用次数或何时使用）：

```
SQL> execute dbms_outln.clear_used(name => 'OUTLINE_FROM_TEXT')

SQL> SELECT used
  2  FROM user_outlines
  3  WHERE name = 'OUTLINE_FROM_TEXT';

USED
------
UNUSED

SQL> SELECT * FROM t WHERE n = 1970;

SQL> SELECT used
  2  FROM user_outlines
  3  WHERE name = 'OUTLINE_FROM_TEXT';

USED
------
USED
```

5. 移动存储概要

Oracle并没有提供用于移动存储概要的特别功能。基本上，必须自己从一个数据字典复制到另一个数据字典。这比较简单，因为数据只保存在了outln模式的三个表中：ol$、ol$hints和ol$nodes。可以使用下面的命令来导入和导出所有可用的存储概要：

```
exp tables=(outln.ol$,outln.ol$hints,outln.ol$nodes) file=outln.dmp

imp full=y ignore=y file=outln.dmp
```

要想移动单个的存储概要（此例中为outline_from_text），可以给export命令添加以下参数：

要想移动一个类别（这里使用test类别）下的所有存储概要，可以给export命令添加以下参数：

```
query="WHERE category='TEST'"
```

请小心，因为根据使用的操作系统和shell，你可能必须添加某些转义字符才能成功传递所有参数。例如，在Linux服务器上，使用bash时，我必须执行以下命令：

```
exp tables=\(outln.ol\$,outln.ol\$hints,outln.ol\$nodes\) file=outln.dmp \
    query=\"WHERE ol_name=\'OUTLINE_FROM_TEXT\'\"
```

6. 编辑存储概要

使用存储概要可以锁定执行计划。然而，只有在查询优化器能够生成高效执行计划并且稍后捕捉到并由存储概要锁定才有用。如果不是这种情况，首先你需要研究的是，为了创建保存高效执行计划的存储概要，是否有可能修改执行环境、访问结构或对象的统计信息。比如，给定SQL语句的执行计划使用了索引扫描，而你想避免使用它，那就应该在测试系统上删除（或隐藏）索引，生成存储概要，然后移动到生产环境中。

当你发现无法强制查询优化器自动生成一个高效的执行计划时，最后的手段是手工修改存储概要。简单地说，你需要修改与存储概要相关联的hint。然而在实践中，你无法对保存在数据字典中的公共存储概要（public stored outline）（这是到目前为止我们讨论的存储概要种类）简单地运行几个SQL

语句。相反，你必须执行像图11-3总结的那样的修改。这个过程是基于私有存储概要（private stored outline）的修改。这些与公共存储概要类似，但不是保存在数据字典中，而是保存在工作表（working table）中。使用工作表的目的就是为了避免直接修改数据字典。因此，要修改存储概要，你需要创建、修改并测试私有存储概要。接着，当私有存储概要工作正常后，就把它改成公共存储概要。Dbms_outln_edit包和CREATE OUTLINE语句的一些扩展都可以修改存储概要。

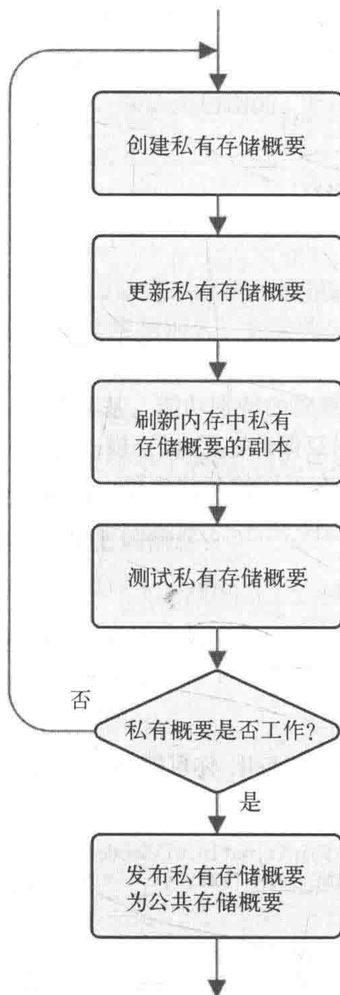


图11-3　修改存储概要期间执行的步骤

根据outline_editing.sql脚本中的例子，我来介绍图11-3总结的整个过程。目的是为以下查询创建和修改存储概要，来用索引扫描替代全表扫描：

```
SQL> EXPLAIN PLAN FOR SELECT * FROM t WHERE n = 1970;

SQL> SELECT * FROM table(dbms_xplan.display(NULL,NULL,'basic'));
```

```
-------------------------------------------
| Id  | Operation                   | Name |
-------------------------------------------
|   0 | SELECT STATEMENT            |      |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |
|   2 |   INDEX RANGE SCAN          | I    |
-------------------------------------------
```

首先，需要创建私有存储概要。因此，会遇到两种情况。第一种情况是像以下SQL语句那样重新创建私有存储概要。PRIVATE关键字指定了要创建的存储概要类型：

```
SQL> CREATE OR REPLACE PRIVATE OUTLINE p_outline_editing
  2  ON SELECT * FROM t WHERE n = 1970;
```

第二种情况是借助于类似以下SQL语句复制已经存在于数据字典中的公共存储概要。PRIVATE和PUBLIC关键字分别指定了需要创建和复制的存储概要类型：

```
SQL> CREATE PRIVATE OUTLINE p_outline_editing FROM PUBLIC outline_editing;
```

两种方法都会在工作表里创建私有存储概要。下面是与存储概要相关的hint列表：

```
SQL> SELECT hint_text
  2  FROM ol$hints
  3  WHERE ol_name = 'P_OUTLINE_EDITING';

HINT_TEXT
-------------------------------------------
INDEX_RS_ASC(@"SEL$1" "T"@"SEL$1" ("T"."N"))
OUTLINE_LEAF(@"SEL$1")
ALL_ROWS
DB_VERSION('11.2.0.3')
OPTIMIZER_FEATURES_ENABLE('11.2.0.3')
IGNORE_OPTIM_EMBEDDED_HINTS
```

一旦创建好私有存储概要，就可以使用常规DML语句修改它。然而，想要修改覆盖所有需求并不是容易的事。一个比较容易实现的办法是再创建一个私有存储概要来复制想要的执行计划，然后交换这两个执行计划的内容。要创建附加存储概要，需要执行以下SQL语句。请注意，hint是用来命令查询使用全表扫描的：

```
SQL> CREATE OR REPLACE PRIVATE OUTLINE p_outline_editing_hinted
  2  ON SELECT /*+ full(t) */ * FROM t WHERE n = 1970;
```

然后通过执行如下SQL语句来交换内容：

```
SQL> UPDATE ol$
  2  SET hintcount = (SELECT hintcount
  3                   FROM ol$
  4                   WHERE ol_name = 'P_OUTLINE_EDITING_HINTED')
  5  WHERE ol_name = 'P_OUTLINE_EDITING';

SQL> DELETE ol$hints
  2  WHERE ol_name = 'P_OUTLINE_EDITING';

SQL> UPDATE ol$hints
  2  SET ol_name = 'P_OUTLINE_EDITING'
  3  WHERE ol_name = 'P_OUTLINE_EDITING_HINTED';
```

下面是交换完后与私有存储概要相关联的hint列表。唯一的不同就是index hint被替换成了full hint：

```
SQL> SELECT hint_text
  2  FROM ol$hints
  3  WHERE ol_name = 'P_OUTLINE_EDITING';

HINT_TEXT
-------------------------------------
FULL(@"SEL$1" "T"@"SEL$1")
OUTLINE_LEAF(@"SEL$1")
ALL_ROWS
DB_VERSION('11.2.0.3')
OPTIMIZER_FEATURES_ENABLE('11.2.0.3')
IGNORE_OPTIM_EMBEDDED_HINTS
```

为了确保内存中的存储概要同步修改，可以执行以下PL/SQL调用：

```
SQL> execute dbms_outln_edit.refresh_private_outline('P_OUTLINE_EDITING')
```

接着，将初始化参数use_private_outlines设置为TRUE或指定私有存储概要所属的类别名来激活和测试私有存储概要。请注意执行计划里的全表扫描和Note部分里的信息，它们都确认了使用私有存储概要。例如：

```
SQL> ALTER SESSION SET use_private_outlines = TRUE;

SQL> EXPLAIN PLAN FOR SELECT * FROM t WHERE n = 1970;

SQL> SELECT * FROM table(dbms_xplan.display(NULL,NULL,'basic +note'));

----------------------------------
| Id  | Operation          | Name |
----------------------------------
|   0 | SELECT STATEMENT   |      |
|   1 |  TABLE ACCESS FULL| T    |
----------------------------------

Note
-----
   - outline "P_OUTLINE_EDITING" used for this statement
```

一旦你满意现有的私有存储概要，就可以使用以下SQL语句将它当作公共存储概要进行发布：

```
SQL> CREATE PUBLIC OUTLINE outline_editing FROM PRIVATE p_outline_editing;
```

7. 删除存储概要

使用DROP OUTLINE语句或dbms_outln包下的drop_by_cat过程，可以删除存储概要。前者删除单个存储概要，而后者删除一个类别下的所有存储概要：

```
DROP OUTLINE outline_from_text

execute dbms_outln.drop_by_cat(cat => 'TEST')
```

要删除私有存储概要，必须使用DROP PRIVATE OUTLINE语句。

8. 权限

创建、修改和删除存储概要需要的系统权限分别是create any outline、alter any outline和drop

any outline。对存储概要来说不存在对象权限。

默认情况下，只有拥有dba或execute_catalog_role角色的用户才能执行dbms_outln包。相反，所有用户都可以执行dbms_outln_edit包下的程序（已将execute权限赋予public）。

终端用户不需要特定权限也可以使用存储概要。

提示 你永远不需要使用outln账户登录。因此，出于安全考虑，应该锁定该帐户或修改默认密码。这很重要，因为该帐户拥有一个非常危险的系统权限：execute any procedure。

11.5.2 何时使用

有两种情况需要考虑使用存储概要。第一，想要优化一条SQL语句而不能在应用里修改它时（例如，无法添加hint）。第二，遇到任何原因导致的执行计划不稳定时。由于存储概要的目的是强制查询优化器为给定SQL语句选择指定执行计划，因此只有当你想明确限制查询优化器选择单个执行计划时才会使用该技巧。

从11.1版本之后，存储概要不支持SQL计划管理。因此，从11.1版本起，只会在标准版里使用存储概要。

11.5.3 陷阱和谬误

奇怪的是，不能在初始化文件（init.ora或spfile.ora）中指定初始化参数use_stored_outlines。因此，必须在每次实例启动后在系统级别设置该参数，或每次在会话建立后在会话级别设置该参数。这两种情况都可以通过数据库触发器来设置初始化参数。例如，下面的触发器仅为名称为Joze的用户设置初始化参数use_stored_outlines：

```
CREATE OR REPLACE TRIGGER enable_outlines AFTER LOGON ON joze.SCHEMA
BEGIN
  EXECUTE IMMEDIATE 'ALTER SESSION SET use_stored_outlines = TRUE';
END;
```

即使在执行计划生成期间应用了存储概要，也并不意味着查询优化器真正选择的就是应用期望生成的执行计划。这个很让人困惑。更何况因为dbms_xplan包的输出和v$sql视图的outline_category列显示了解析阶段使用的存储概要。下面的例子，是outline_unreproducible.sql脚本生成的输出节选：

```
SQL> SELECT * FROM t WHERE n = 1970;

-----------------------------------------------------
| Id  | Operation                          | Name |
-----------------------------------------------------
|   0 | SELECT STATEMENT                   |      |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| T    |
|*  2 |   INDEX RANGE SCAN                 | I    |
-----------------------------------------------------

   2 - access("N"=1970)
```

```
Note
-----
   - outline "OUTLINE_UNREPRODUCIBLE" used for this statement

SQL> DROP INDEX i;

SQL> SELECT * FROM t WHERE n = 1970;

---------------------------------
| Id  | Operation          | Name |
---------------------------------
|   0 | SELECT STATEMENT   |      |
|*  1 |  TABLE ACCESS FULL| T    |
---------------------------------

   1 - filter("N"=1970)

Note
-----
   - outline "OUTLINE_UNREPRODUCIBLE " used for this statement
```

存储概要最重要的一个属性是，它们是从代码中分离出来的。不过这也会导致问题。实际上，由于存储概要与SQL语句之间没有直接的引用，开发人员完全可以忽略存储概要的存在。因此，如果开发人员修改了SQL语句而导致它的签名改变，存储概要也就跟着失效了。同理，当你要部署的应用需要使用存储概要来保证正常运行时，在数据库安装期间别忘了安装它们。

需要注意的是，当存储概要依赖的对象被删除时，存储概要不会被删除。这并不是个问题。例如，如果一个表或索引由于必须重组或移动而需要重建，那么存储概要没有被删除就是好事；否则还需要重建它们。

两个有相同文本的SQL语句拥有相同的签名。即使它们引用的对象在不同的模式下。这代表单个存储概要可以被两个同名但处于不同模式中的表使用。再次强调，你需要特别小心，尤其是数据库里同样的对象有多个副本时。

当某个SQL语句有存储概要，同时还有SQL配置文件和/或SQL计划基准（plan baseline）时，查询优化器仅会使用存储概要。当然，前提是仅当存储概要的使用处于活动状态时。

11.6　SQL 配置文件

你可以将SQL优化委派给称为自动调整优化器（Automatic Tuning Optimizer）的查询优化器的一个组件。将此任务委派给在第一个位置无法找到有效执行计划的同一个组件，这可能看起来很奇怪。但实际上，这两种情况很不同。事实上，在正常情况下，由于查询优化器需要快速运转（基本是亚秒级）而被迫生成次优的执行计划。相反，自动调整优化器会有更多的时间来执行一个高效的执行计划。进一步讲，它可以使用耗时技术（如模拟分析）和加大动态采样技术的使用率来验证它的估算。

自动调整优化器是通过SQL优化顾问（SQL Tuning Advisor）引入的。它的目的是分析SQL语句并针对其性能的提高提出建议，包括收集缺失或陈旧的统计信息，创建新的索引，修改SQL语句或使用SQL配置文件。接下来的部分会专门介绍SQL配置文件。

关于SQL配置文件必须要知道的是它只可以通过SQL优化顾问生成。但在稍后我会介绍，你也可以手工创建它们。

11.6.1 工作原理

接下来的几个部分会介绍什么是SQL配置文件以及如何使用它们，还会提供关于它们的内部工作的信息。要管理SQL配置文件，可以使用集成到企业管理器（Enterprise Manager）的图形界面。我们不会花时间在这上面，因为在我看来，如果你懂得后台发生了什么，那么使用图形界面就不会有问题。

1. SQL配置文件的定义

SQL配置文件是一种对象，这种对象包含可帮助查询优化器为特定SQL语句找到高效执行计划的信息。SQL配置文件提供关于以下各项的信息：执行环境、对象统计信息和与查询优化器执行的评估相关的更正。SQL配置文件的主要优势之一是，可以影响查询优化器而不用修改SQL语句或它所在会话的执行环境。换句话说，它对于连接到数据库引擎的应用是透明的。要理解SQL配置文件是如何工作的，让我们来看看它是如何生成和使用的。

图11-4举例说明SQL配置文件生成期间的执行步骤。简单来说，用户请求SQL优化顾问来优化SQL语句，然后当SQL配置文件提出建议后，SQL优化顾问就会接受。

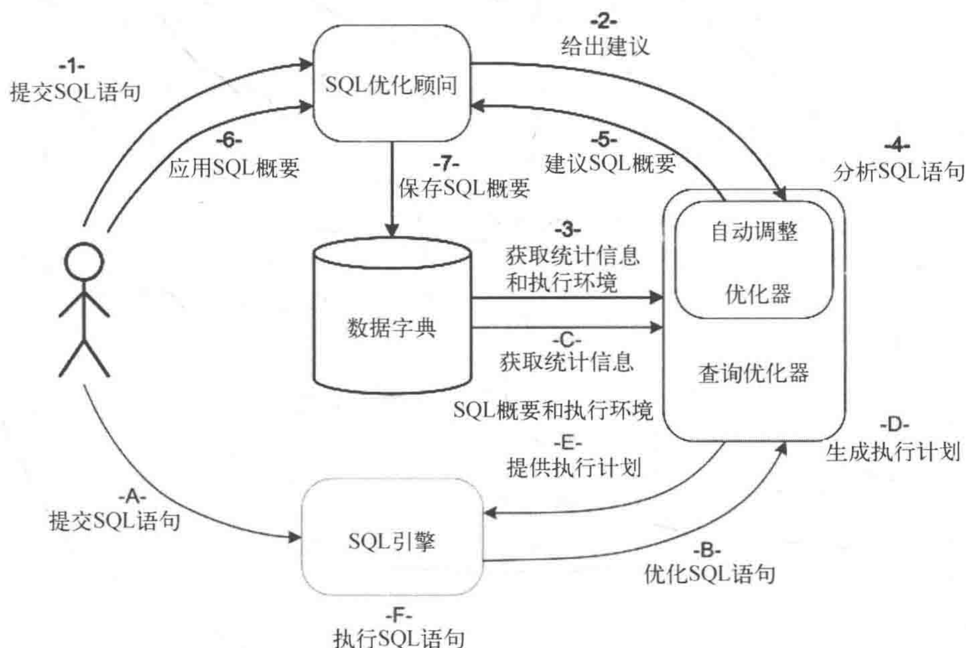


图11-4 SQL配置文件生成期间执行的步骤

下面是详细的步骤。

(1) 用户将性能糟糕的SQL语句传递给SQL优化顾问。

(2) SQL优化顾问要求自动调整优化器针对需要优化的SQL语句给出建议。

(3) 查询优化器获取系统统计信息、与SQL语句引用的对象相关的对象统计信息以及设置执行环境的初始化参数。

(4) 分析SQL语句。在这个过程中，自动调整优化器执行分析并部分执行SQL语句来验证它的猜测。

(5) 自动调整优化器将SQL配置文件返回给SQL优化顾问。

(6) 用户使用SQL配置文件。

(7) 将SQL配置文件存储到数据字典中。

图11-5举例说明使用SQL配置文件期间执行的步骤。重点是整个过程对用户来说是透明的。

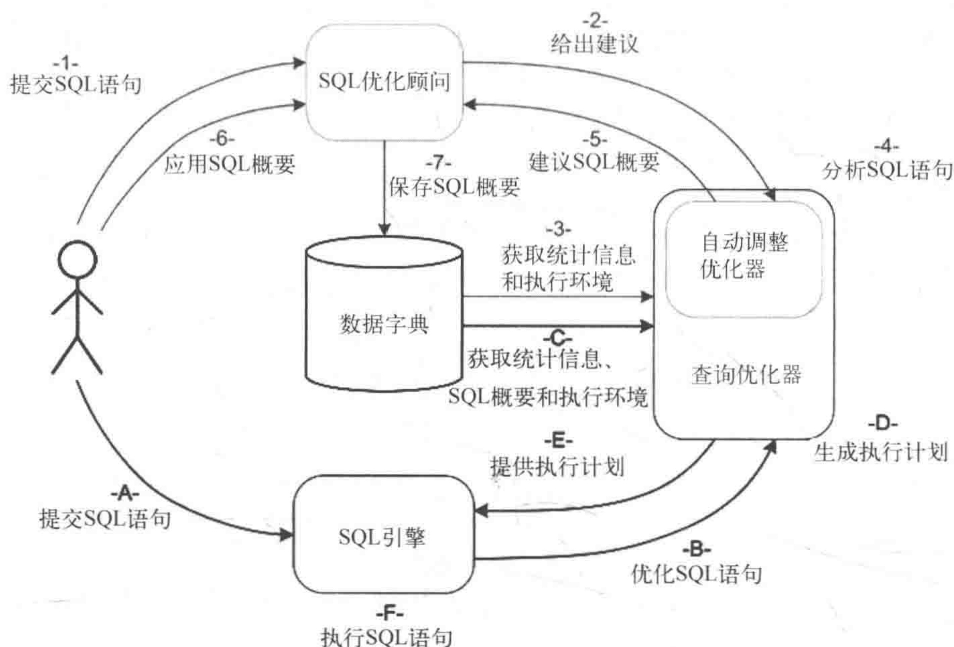


图11-5　SQL语句执行期间执行的主要步骤

下面是详细的步骤。

A. 用户将SQL语句发送给SQL引擎来执行。

B. SQL引擎要求查询优化器提供执行计划。

C. 查询优化器获取系统统计信息、与SQL语句引用的对象相关的对象统计信息、SQL配置文件以及设置执行环境的初始化参数。

D. 查询优化器分析SQL语句并生成执行计划。

E. 将执行计划传递给SQL引擎。

F. SQL引擎执行SQL语句。

下一部分详细介绍SQL配置文件生成和使用期间执行的核心步骤。特别关注涉及用户的步骤。让我们先从SQL优化顾问开始。

2. SQL优化顾问

通过dbms_sqltune包可以访问SQL优化顾问的核心界面。此外，在企业管理器中还集成了一个图形界面。通过这两个界面可以执行优化任务（tuning task），还可以查看产生的建议并接受建议。在这里我并不会向你展示图形用户界面如何工作，因为更重要的是要了解后台发生了什么。

注意 要使用SQL优化顾问和dbms_sqltune包，必须获得使用Diagnostics Pack和Tuning Pack的许可。记住，这些选件仅在企业版可用。

要启动优化任务，必须调用dbms_sqltune包中的create_tuning_task函数，并将以下各项之一作为一个参数传递（函数会重载四次以接受不同类型的参数）。

- SQL语句的文本。
- 对存储在库缓存中的SQL语句的引用（sql_id）。
- 对存储在AWR（Automatic Workload Repository，自动工作负载存储库）中的SQL语句的引用（sql_id）。
- SQL优化集的名称。

SQL优化集（SQL TUNING SETS）

简单地说，**SQL优化集**是将一组SQL语句与其关联的执行环境、执行统计信息和可选执行计划存储到一起的对象。SQL优化集是使用dbms_sqltune包来管理的。

需要Tuning Pack或Real Application Testing才能使用SQL优化集，也就是说要使用企业版。

可以在*Oracle Database Performance Tuning Guide*手册（11.2及之后版本）或者*Oracle Database SQL Tuning Guide*手册（12.1及之后版本）中找到关于SQL优化集的更多信息。

为了通过将单个SQL语句当作一个参数来简化dbms_sqltune包中的create_tuning_task函数的执行，我编写了tune_last_statement.sql脚本。其想法是你执行希望已在SQL*Plus中分析过的SQL语句，然后不使用参数来调用该脚本。该脚本会从v$session视图中获取当前会话执行的最后一条SQL语句的引用（sql_id），然后创建并执行一个引用该脚本的优化任务。该脚本的核心部分为以下匿名PL/SQL块：

```
DECLARE
  l_sql_id v$session.prev_sql_id%TYPE;
BEGIN
  SELECT prev_sql_id INTO l_sql_id
  FROM v$session
  WHERE audsid = sys_context('userenv','sessionid');

  :tuning_task := dbms_sqltune.create_tuning_task(sql_id => l_sql_id);
  dbms_sqltune.execute_tuning_task(:tuning_task);
END;
```

优化任务会将多个数据字典视图中的分析输出具体化。可以使用dbms_sqltune包中的report_tuning_task函数来生成关于分析的详细报告，而不用直接查询视图。下面的查询展示了它的使用。请

注意需要使用上一个PL/SQL块返回的优化任务名称来引用优化任务：

```
SELECT dbms_sqltune.report_tuning_task(:tuning_task)
FROM dual
```

上个查询会生成类似以下的报告来建议使用SQL配置文件。请注意这部分选自profile_opt_estimate.sql脚本生成的输出。第一部分显示分析和SQL语句的基本信息。第二部分显示结果和建议。本例中，建议使用SQL配置文件。最后一部分显示应用建议之前和之后的执行计划：

```
GENERAL INFORMATION SECTION
-------------------------------------------------------------------------------
Tuning Task Name   : TASK_3401
Tuning Task Owner  : CHRIS
Workload Type      : Single SQL Statement
Scope              : COMPREHENSIVE
Time Limit(seconds): 42
Completion Status  : COMPLETED
Started at         : 08/02/2013 15:31:44
Completed at       : 08/02/2013 15:31:45

-------------------------------------------------------------------------------
Schema Name: CHRIS
SQL ID     : bczb6dmm8gcfs
SQL Text   : SELECT * FROM t1, t2 WHERE t1.col1 = 666 AND t1.col2 > 42 AND
             t1.id = t2.id

-------------------------------------------------------------------------------
FINDINGS SECTION (1 finding)
-------------------------------------------------------------------------------

1- SQL Profile Finding (see explain plans section below)
--------------------------------------------------------
  A potentially better execution plan was found for this statement.

  Recommendation (estimated benefit: 65.35%)
  ------------------------------------------
  - Consider accepting the recommended SQL profile.
    execute dbms_sqltune.accept_sql_profile(task_name => 'TASK_3401',
            task_owner => 'CHRIS', replace => TRUE);

-------------------------------------------------------------------------------
EXPLAIN PLANS SECTION
-------------------------------------------------------------------------------

1- Original With Adjusted Cost
------------------------------
Plan hash value: 2452363886

---------------------------------------------------------------------------------------
| Id  | Operation                  | Name           | Rows | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |                | 5000 |  9892K|   6210 (1)| 00:01:40 |
|   1 |  NESTED LOOPS              |                |      |       |            |          |
|   2 |   NESTED LOOPS             |                | 5000 |  9892K|   6210 (1)| 00:01:40 |
```

```
|   3 |    TABLE ACCESS BY INDEX ROWID| T1           | 9646 |  9542K|    1385 (0)| 00:00:23 |
|*  4 |     INDEX RANGE SCAN          | T1_COL1_COL2_I | 9500 |       |      27 (0)| 00:00:01 |
|*  5 |    INDEX UNIQUE SCAN          | T2_PK        |    1 |       |       0 (0)| 00:00:01 |
|   6 |   TABLE ACCESS BY INDEX ROWID | T2           |    1 |  1013 |       1 (0)| 00:00:01 |
---------------------------------------------------------------------------------------------

  4 - access("T1"."COL1"=666 AND "T1"."COL2">42 AND "T1"."COL2" IS NOT NULL)
  5 - access("T1"."ID"="T2"."ID")

2- Using SQL Profile
--------------------
Plan hash value: 2959412835

---------------------------------------------------------------------------------
| Id  | Operation           | Name | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
---------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      | 5000  |  9892K|       |   1081 (1)| 00:00:18 |
|*  1 |  HASH JOIN          |      | 5000  |  9892K|  5008K|   1081 (1)| 00:00:18 |
|   2 |   TABLE ACCESS FULL| T2   | 5000  |  4946K|       |    174 (0)| 00:00:03 |
|*  3 |   TABLE ACCESS FULL| T1   | 9646  |  9542K|       |    344 (0)| 00:00:06 |
---------------------------------------------------------------------------------

  1 - access("T1"."ID"="T2"."ID")
  3 - filter("T1"."COL1"=666 AND "T1"."COL2">42)
```

要使用SQL优化顾问推荐的SQL配置文件，你需要应用它。下一部分会介绍如何应用。无论是否应用SQL配置文件，一旦不再需要优化任务，就可以调用dbms_sqltune包中的drop_tuning_task过程来删掉它：

```
dbms_sqltune.drop_tuning_task('TASK_3401');
```

3. 接受SQL配置文件

dbms_sqltune包中的accept_sql_profile过程用来接受SQL优化顾问建议的SQL配置文件。它接受以下参数。

- Task_name和task_owner参数引用建议SQL配置文件的优化任务。
- Name和description参数指定SQL配置文件的名称和描述。例如，使用生成它的脚本名作为它的名称。
- Category参数用于将几个SQL配置文件组合起来，以便于管理。默认值为DEFAULT。
- Replace参数指定是否替换已经可用的SQL配置文件。默认值为FALSE。
- Force_match参数指定如何执行文本标准化。默认值是FALSE。下一部分会给出更多关于文本标准化的信息。

只有task_name是强制性参数。例如，要应用上面报告中推荐的SQL配置文件，你需要使用以下PL/SQL调用：

```
dbms_sqltune.accept_sql_profile(task_name   => 'TASK_3401',
                                task_owner  => user,
                                name        => 'opt_estimate',
                                description => NULL,
                                category    => 'TEST',
```

```
                                  force_match => TRUE,
                                  replace     => TRUE);
```

一旦应用，SQL配置文件就会保存在数据字典中。dba_sql_profiles视图显示了它的信息。此外，从12.1版本之后，cdb_sql_profiles视图也可用。由于SQL配置文件不会被绑定到特定用户，因此all_sql_profiles和user_sql_profiles视图不存在：

```
SQL> SELECT category, sql_text, force_matching
  2  FROM dba_sql_profiles
  3  WHERE name = 'opt_estimate';

CATEGORY SQL_TEXT                                      FORCE_MATCHING
-------- --------------------------------------------- --------------
TEST     SELECT * FROM t1, t2 WHERE t1.col1 = 666 AND  YES
         t1.col2 > 42 AND t1.id = t2.id
```

accept_sql_profile函数与accept_sql_profile过程一样。唯一不同的是函数会返回SQL配置文件名。如果没有在输入函数中指定名称而系统需要生成结果时，这会变得很有用。

4. 修改SQL配置文件

创建SQL配置文件之后，可以使用dbms_sqltune包中的alter_sql_profile过程来修改它的一些属性，并且还可以使用它来修改SQL配置文件的状态（enabled或disabled）。该过程接受以下参数。

- Name参数指定要修改的SQL配置文件。
- Attribute_name参数指定要修改的属性。它能接受的值有：name、description、category和status。
- Value参数指定新的属性值。

这三个参数是强制的。例如，以下PL/SQL调用会禁用上面例子创建的SQL配置文件：

```
dbms_sqltune.alter_sql_profile(name           => 'opt_estimate',
                               attribute_name => 'status',
                               value          => 'disabled');
```

5. 文本标准化

SQL配置文件的一个主要优势是尽管它应用于某个SQL语句，但它并不会对SQL语句做任何修改。实际上，SQL配置文件保存在数据字典中，并且查询优化器会自动选择它们。图11-6显示了在选择过程中会实施的基本步骤。首先，会使SQL语句标准化，这代表不仅要不区分大小写，还要不使用空格。基于结果的SQL语句会计算出签名。然后会根据签名在数据字典中进行查找。每当找到有相同签名的SQL配置文件时，就会执行检查来确保SQL语句是最优的并且关联SQL配置文件的SQL语句也是等价的。这一步很重要，因为签名其实是个散列值，因此可能会存在冲突。如果检测成功，会将与SQL配置文件关联的hint加入到生成的执行计划中。

如果SQL语句包含的文字发生改变，它会像散列值签名一样改变。因此，SQL配置文件是没用的，因为SQL配置文件被绑定到某个仅会执行一次的SQL语句。为了避免这个问题，数据库引擎会在标准化阶段去除文字部分。要启用这个功能，需要在应用SQL配置文件时将force_match参数设置为TRUE。

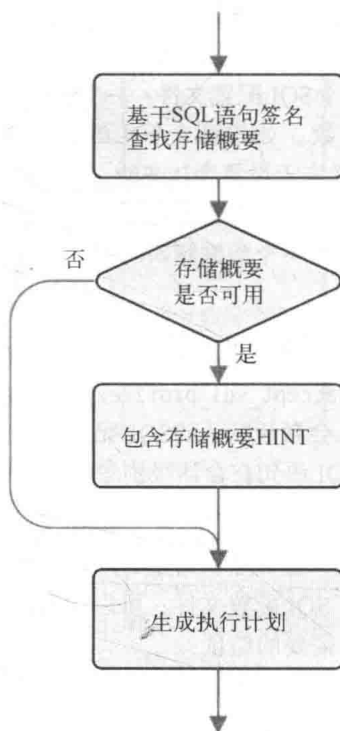


图11-6 SQL配置文件选择期间实施的主要步骤

为了研究文本标准化的工作原理，可以使用dbms_sqltune包中的sqltext_to_signature函数。它需要两个输入参数，sql_text和force_match。前者指定SQL语句，后者指定文本标准化的类型。下面是节选自profile_signature.sql脚本生成的输出，展示了在签名不同但是相似的SQL语句上force_match参数的影响。

- force_match设置为FALSE：空格和不区分大小写。

```
SQL_TEXT                                                     SIGNATURE
--------------------------------------------------- --------------------
SELECT * FROM dual  WHERE dummy = 'X'                 7181225531830258335
select  *  from  dual  where  dummy='X'               7181225531830258335
SELECT * FROM dual WHERE dummy  = 'x'                18443846411346672783
SELECT * FROM dual WHERE dummy  = 'Y'                  909903071561515954
SELECT * FROM dual WHERE dummy  = 'X' OR dummy = :b1 14508885911807130242
SELECT * FROM dual WHERE dummy  = 'Y' OR dummy = :b1   816238779370039768
```

- force_match设置为TRUE：空格和不区分大小写和文字。然而，如果SQL语句中使用了绑定变量，那么不会执行文字替换。

```
SQL_TEXT                                                     SIGNATURE
--------------------------------------------------- --------------------
SELECT * FROM dual WHERE dummy = 'X'                10668153635715970930
select  *  from  dual  where  dummy='X'             10668153635715970930
SELECT * FROM dual WHERE dummy  = 'x'                10668153635715970930
SELECT * FROM dual WHERE dummy  = 'Y'                10668153635715970930
```

```
SELECT * FROM dual WHERE dummy  = 'X' OR dummy = :b1 14508885911807130242
SELECT * FROM dual WHERE dummy  = 'Y' OR dummy = :b1   816238779370039768
```

请注意，同一SQL语句可以有两个SQL配置文件：一个使用设置为FALSE的force_match参数，另一个使用设置为TRUE的force_match参数。如果两个SQL配置文件都存在，那么使用设置为FALSE的force_match参数的SQL配置文件会优先于设置为TRUE的。这是因为设置为FALSE的force_match参数会比另一个更详细些。这表示可以使用一个SQL配置文件来对应大多数文字，而另一个来对应需要特别处理的（比如，当对文字的限制应用在一个数据倾斜的列上时）。

6. 激活SQL配置文件

可以在系统或会话级别通过初始化参数sqltune_category来控制SQL配置文件的激活。默认值为DEFAULT。这也是dbms_sqltune包中的accept_sql_profile过程中category参数的默认值。因此，如果应用SQL配置文件时未指定类别，那么会激活默认的SQL配置文件。应用SQL配置文件时，会把类别名当作一个值来处理。比如，下面的SQL语句在会话级别激活属于test类别的SQL配置文件：

```
ALTER SESSION SET sqltune_category = test
```

这个初始化参数只支持单个类别。很显然在给定时间内一个会话只能激活一个类别。

为了查明查询优化器是否使用了SQL配置文件，可以利用dbms_xplan包中的函数。正如下面的例子，它们输出的Note部分明确给出了需要的信息：

```
SQL> EXPLAIN PLAN FOR SELECT * FROM t ORDER BY id;

SQL> SELECT * FROM table(dbms_xplan.display);

-------------------------------------------
| Id  | Operation                   | Name |
-------------------------------------------
|   0 | SELECT STATEMENT            |      |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |
|   2 |   INDEX FULL SCAN           | T_PK |
-------------------------------------------

Note
-----
  - SQL profile "import_sql_profile" used for this statement
```

对于保存在库缓存中的游标，v$sql视图的sql_profile列显示了在游标执行计划生成期间使用的SQL配置文件名。当没有使用SQL配置文件时，该列值为NULL。

7. 移动SQL配置文件

dbms_sqltune包提供了多个过程以在数据库之间移动SQL配置文件。如图11-7显示，会提供以下功能。

- 可以通过create_stgtab_sqlprof过程创建临时表。
- 可以通过pack_stgtab_sqlprof过程将数据字典中的SQL配置文件复制到临时表中。
- 可以通过remap_stgtab_sqlprof过程修改保存在临时表中的SQL配置文件名和类别。
- 可以通过unpack_stgtab_sqlprof过程将临时表中的SQL配置文件复制到数据字典中。

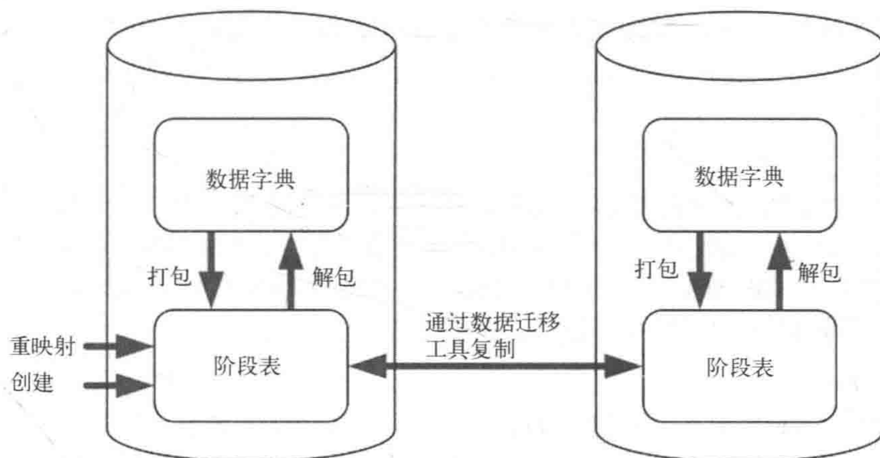


图11-7 使用dbms_sqltune包移动SQL配置文件

请注意，在数据库之间移动临时表依靠的是数据移动技术（例如，数据泵（Data Pump）或旧有的导出（export）和导入（import）程序），并不依靠dbms_sqltune包本身。

下面的例子，引用自profile_cloning.sql脚本，显示了如何在单个数据库中复制SQL配置文件。首先，mystgtab临时表是在当前模式（schema）中创建的：

```
dbms_sqltune.create_stgtab_sqlprof(table_name      => 'MYSTGTAB',
                                   schema_name     => user,
                                   tablespace_name => 'USERS');
```

接着，会将名称为opt_estimate的SQL配置文件从数据字典复制到临时表中：

```
dbms_sqltune.pack_stgtab_sqlprof(profile_name         => 'opt_estimate',
                                 profile_category     => 'TEST',
                                 staging_table_name   => 'MYSTGTAB',
                                 staging_schema_owner => user);
```

将SQL配置文件复制回数据字典中之前，必须修改SQL配置文件名。同时，也要修改它的类别：

```
dbms_sqltune.remap_stgtab_sqlprof(old_profile_name     => 'opt_estimate',
                                  new_profile_name     => 'opt_estimate_clone',
                                  new_profile_category => 'TEST_CLONE',
                                  staging_table_name   => 'MYSTGTAB',
                                  staging_schema_owner => user);
```

最后，将SQL配置文件从临时表复制到数据字典中。由于参数会被替换成TRUE，同名的SQL配置文件也会被改写：

```
dbms_sqltune.unpack_stgtab_sqlprof(profile_name         => 'opt_estimate_clone',
                                   profile_category     => 'TEST_CLONE',
                                   replace              => TRUE,
                                   staging_table_name   => 'MYSTGTAB',
                                   staging_schema_owner => user);
```

8. 删除SQL配置文件

可以使用dbms_sqltune包中的drop_sql_profile过程来删除数据字典中的SQL配置文件。Name参数

指定SQL配置文件名。Ignore参数指定当SQL配置文件不存在时是否报错。默认值为FALSE：

```
dbms_sqltune.drop_sql_profile(name   => 'opt_estimate',
                              ignore => TRUE);
```

9. 权限

要创建、修改和删除SQL配置文件，分别需要create any sql profile、alter any sql profile和drop any sql profile系统权限。然而，从11.1版本开始，这三个系统权限不再支持administer sql management系统权限对象。SQL配置文件没有对象权限。要使用SQL优化顾问，就需要advisor系统权限。

最终用户不需要特定权限也可以使用SQL配置文件。

10. 未公开特性

SQL配置文件如何影响查询优化器？Oracle并未在其文档中给出答案。我认为高效地使用特性的最好方法就是了解它的工作原理。因此，让我们来看看它的内部。简单地说，SQL配置文件保存了一组hint来表示查询优化器执行的优化。其中一些hint是在文档里有记录的，并且也用于其他环境。其他hint是未公开的，并且通常只会由SQL配置文件使用。换句话说，它们都为了这个目的而使用。它们全部都是普通的hint，因此也可以直接加入到SQL语句中。

在讨论如何查询与SQL配置文件关联的hint列表前，让我们先引入一个基于profile_all_rows.sql脚本的例子。它的目的是为了展示，使用SQL配置文件是可以命令查询优化器改变优化模式的。在这个特定的例子中，需要改变优化器模式，是因为查询包含了rule hint，这会强制查询优化器在基于规则的模式下工作。查询和它的执行计划如下：

```
SQL> SELECT /*+ rule */ * FROM t ORDER BY id;
---------------------------------------------
| Id  | Operation                    | Name |
---------------------------------------------
|   0 | SELECT STATEMENT             |      |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |
|   2 |   INDEX FULL SCAN            | T_PK |
---------------------------------------------

Note
-----
   - rule based optimizer used (consider using cbo)
```

在让SQL优化顾问在查询上工作并且应用SQL配置文件后，执行计划也会改变。正如Note部分指出的那样，会在执行计划生成期间使用SQL配置文件：

```
SQL> SELECT /*+ rule */ * FROM t ORDER BY id;
-----------------------------------------------------------------------------------
| Id  | Operation           | Name | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      | 10000 |  1015K|       |   277   (1)| 00:00:04 |
|   1 |  SORT ORDER BY      |      | 10000 |  1015K|  1120K|   277   (1)| 00:00:04 |
|   2 |   TABLE ACCESS FULL| T    | 10000 |  1015K|       |    38   (0)| 00:00:01 |
-----------------------------------------------------------------------------------

Note
-----
   - SQL profile "all_rows" used for this statement
```

不幸的是，与SQL配置文件关联的hint无法通过数据字典视图显示。实际上，只有两个视图能提供关于SQL配置文件的信息：dba_sql_profiles视图和在12.1多租户环境下的cdb_sql_profiles视图，它们提供除了hint以外的所有信息。如果想知道哪些hint被用于SQL配置文件，那么你有两个选择。第一个是直接查询内部数据字典表。下面的查询会介绍针对由profile_all_rows.sql脚本生成的SQL配置文件该如何查询。请注意，会用到两个初始化参数hint（all_rows和optimizer_features_enable）。此外，要命令查询优化器忽略当前SQL语句中的hint（本例是rule hint），会用到ignore_optim_embedded_hints。

- 该查询在10.2版本中可用：

```
SQL> SELECT attr_val
  2  FROM sys.sqlprof$ p, sys.sqlprof$attr a
  3  WHERE p.sp_name = 'all_rows'
  4  AND p.signature = a.signature
  5  AND p.category = a.category;

ATTR_VAL
-----------------------------------
ALL_ROWS
OPTIMIZER_FEATURES_ENABLE(default)
IGNORE_OPTIM_EMBEDDED_HINTS
```

- 该查询在11.1版本中可用：

```
SQL> SELECT extractValue(value(h),'.') AS hint
  2  FROM sys.sqlobj$data od, sys.sqlobj$ so,
  3       table(xmlsequence(extract(xmltype(od.comp_data),'/outline_data/hint'))) h
  4  WHERE so.name = 'all_rows'
  5  AND so.signature = od.signature
  6  AND so.category = od.category
  7  AND so.obj_type = od.obj_type
  8  AND so.plan_id = od.plan_id;

HINT
-----------------------------------
ALL_ROWS
OPTIMIZER_FEATURES_ENABLE(default)
IGNORE_OPTIM_EMBEDDED_HINTS
```

第二种可能是将SQL配置文件移动到临时表中，这在“移动SQL配置文件”部分介绍过。接着，使用类似以下的查询，可以从临时表中获取hint。请注意会执行通过table函数的非嵌套查询，因为hint存储在VARCHAR2变长数组中：

```
SQL> SELECT *
  2  FROM table(SELECT attributes
  3             FROM mystgtab
  4             WHERE profile_name = 'opt_estimate');

COLUMN_VALUE
-----------------------------------
ALL_ROWS
OPTIMIZER_FEATURES_ENABLE(default)
IGNORE_OPTIM_EMBEDDED_HINTS
```

SQL配置文件不仅可以更改优化器的模式，实际上，它还可以用来校正查询优化器执行错误的基数估算。Profile_opt_estimate.sql脚本展示的就是这样的例子。使用第10章介绍的技巧可以识别错误的估算。可以看到在下面的例子中，几项操作的估算基数（E-Rows）与真实基数（A-Rows）完全不同：

```
-------------------------------------------------------------------------------------
| Id  | Operation                      | Name          | Starts | E-Rows | A-Rows |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |               |      1 |        |   4750 |
|   1 |  NESTED LOOPS                  |               |      1 |        |   4750 |
|   2 |   NESTED LOOPS                 |               |      1 |     20 |   4750 |
|   3 |    TABLE ACCESS BY INDEX ROWID| T1            |      1 |     20 |   9500 |
|*  4 |     INDEX RANGE SCAN           | T1_COL1_COL2_I |      1 |     20 |   9500 |
|*  5 |    INDEX UNIQUE SCAN           | T2_PK         |   9500 |      1 |   4750 |
|   6 |   TABLE ACCESS BY INDEX ROWID  | T2            |   4750 |      1 |   4750 |
-------------------------------------------------------------------------------------
```

如果使用SQL优化顾问来分析这样的案例并且应用它的建议，就像profile_opt_estimate.sql脚本那样，那么会创建包含以下hint的SQL配置文件：

```
OPT_ESTIMATE(@"SEL$1", INDEX_SCAN, "T1"@"SEL$1", "T1_COL1_COL2_I", SCALE_ROWS=477.9096254)
OPT_ESTIMATE(@"SEL$1", NLJ_INDEX_SCAN, "T2"@"SEL$1", ("T1"@"SEL$1"), "T2_PK",
             SCALE_ROWS=0.4814075109)
OPT_ESTIMATE(@"SEL$1", NLJ_INDEX_FILTER, "T2"@"SEL$1", ("T1"@"SEL$1"), "T2_PK",
             SCALE_ROWS=0.4814075109)
OPT_ESTIMATE(@"SEL$1", TABLE, "T1"@"SEL$1", SCALE_ROWS=486.2776343)
OPTIMIZER_FEATURES_ENABLE(default)
```

需要额外注意的是未公开的hint opt_estimate。使用这个特别的hint，就可以通知查询优化器它的一些估算是错误的并且还可以得知错误程度。例如，第一个hint告诉查询优化器对访问表t1的操作估算按比例增加大约478倍（“大约”是因为9500/20的分母在dbms_xplan的输出中被四舍五入了）。

适当地使用SQL配置文件，估算会变得更精确。同样请注意查询优化器选择了另一个执行计划，它是最初用来创建SQL配置文件的：

```
------------------------------------------------------------------
| Id  | Operation          | Name | Starts | E-Rows | A-Rows |
------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |        |   4750 |
|*  1 |  HASH JOIN         |      |      1 |   5000 |   4750 |
|   2 |   TABLE ACCESS FULL| T2   |      1 |   5000 |   5000 |
|*  3 |   TABLE ACCESS FULL| T1   |      1 |   9666 |   9500 |
------------------------------------------------------------------
```

另一个SQL配置文件的用处是当对象存在错误或丢失对象统计信息时。当然这不应该发生，但当它发生并且动态采样无法为查询优化器提供需要的信息时，就可以使用SQL配置文件。Profile_object_stats.sql脚本提供了这样的例子。脚本生成的SQL配置文件是由hint组成的，尤其是以下这些。正如hint名显示的那样，每个hint都在为表、对象或列提供对象统计信息：

```
TABLE_STATS("CHRIS"."T2", scale, blocks=735 rows=5000)
INDEX_STATS("CHRIS"."T2", "T2_PK", scale, blocks=14 index_rows=5000)
COLUMN_STATS("CHRIS"."T2", "PAD", scale, length=1000)
```

```
COLUMN_STATS("CHRIS"."T2", "COL2", scale, length=3)
COLUMN_STATS("CHRIS"."T2", "COL1", scale, length=3)
COLUMN_STATS("CHRIS"."T2", "ID", scale, length=3 distinct=5000 nulls=0 min=2 max=10000)
```

对于这部分关于未公开特性的内容，我最后想介绍的是手工创建SQL配置文件。换句话说，代替询问SQL优化顾问分析并应用SQL配置文件，你可以建立自己的SQL配置文件。通过调用dbms_sqltune包中的import_sql_profile过程来手工创建SQL配置文件。下面的示例是基于profile_import.sql脚本的调用。Sql_text参数指定了绑定SQL配置文件的SQL语句，profile参数指定hint列表。其他参数与之前介绍的accept_sql_profile过程的参数具有相同的定义：

```
dbms_sqltune.import_sql_profile(
  name        => 'import_sql_profile',
  description => 'SQL profile created manually',
  category    => 'TEST',
  sql_text    => 'SELECT * FROM t ORDER BY id',
  profile     => sqlprof_attr('first_rows(42)','optimizer_features_enable(default)'),
  replace     => FALSE,
  force_match => FALSE
);
```

注意 尽管dbms_sqltune包中的import_sql_profile并不是官方记录的，但创建SQL配置文件的方法，与应用SQL优化顾问的建议而由数据库引擎创建的SQL配置文件是一样的。因此，我认为使用import_sql_profile过程没问题。此外，在Oracle Suport 说明*SQLT (SQLTXPLAIN) - Tool that helps to diagnose a SQL statement performing poorly or one that produces wrong results* (215187.1)中的coe_xfr_sql_profile.sql脚本使用了同样的过程来创建SQL配置文件。另外，可以执行coe_xfr_sql_profile.sql脚本来为库缓存中缓存的或AWR中存储的游标创建SQL配置文件。

11.6.2 何时使用

每当要优化一条特定的SQL语句并且无法在应用中更改它（例如，无法添加hint）时，都应该考虑使用SQL配置文件。请记住，SQL配置文件的目的是为查询优化器提供关于要处理的数据以及关于执行环境的附加信息。因此，不要在需要为某条特定SQL语句强制使用某个特定执行计划时使用该技术。

为此，应该使用存储概要或SQL计划管理。唯一的例外是当你想要利用与force_match参数相关的文本标准化功能时。实际上，存储概要和SQL计划管理都不提供类似的功能。

将初始化参数control_management_pack_access设置为none或diagnostic时，将无法使用SQL优化顾问。如果尝试去使用，那么数据库引擎会引发ORA-13717: Tuning Package License is needed for using this feature错误。此外，查询优化器会忽略现有SQL配置文件。

11.6.3 陷阱和谬误

SQL配置文件最重要的属性之一是，它们与代码是分开的。然而这也会带来问题。实际上，由于在SQL配置文件与SQL语句之间没有直接的关联，开发人员很可能会彻底忽略SQL配置文件的存在。

结果，如果开发人员修改SQL语句将会导致它的签名发生改变，这样SQL配置文件就不会再生效了。同样，当你部署一个应用需要依靠SQL配置文件来保证它执行正确时，必须记得在数据库设置期间安装它们。

如果需要生成SQL配置文件，最好的做法是在生产环境中生成（如果可行），然后移动到其他环境中去做测试。但是，问题是在移动SQL配置文件之前，你不得不应用它。你不会想在未测试之前就在生产环境中应用它，因此需要确保应用的SQL配置文件使用的类别与初始化参数sqltune_category激活的类别不同。那样，SQL配置文件就不会在生产数据库上使用。总之，你总是可以在过后修改SQL配置文件的类别。

需要注意的是，SQL配置文件依赖的对象被删除时，SQL配置文件并不会被删除。但这并不是问题。例如，如果一个表或索引因为它必须重组或移动而需要重建，那么SQL配置文件没被删除就是好事。否则，就有必要重建它们。

两个有相同文本的SQL语句拥有相同的签名。即使它们引用的对象在不同的用户下。这代表单个存储概要可以被两个同名但是不同用户的表使用。再次强调，你需要小心，尤其是当数据库中同样的对象有多个副本时。

在11.2.0.2及之前的版本中，因为Oracle Support文档*SQL profile not used in the Active Physical Standby*（10050057.8）中描述的bug，导致SQL配置文件在Active Data Guard环境下受限。你可以在主实例上使用SQL配置文件，但并不总是能在备用实例上使用。

当SQL语句有SQL配置文件和存储概要时，查询优化器会仅使用存储概要。当然，前提是存储概要处于激活状态。

当SQL语句有SQL配置文件和SQL计划基线时，查询优化器会尝试合并与SQL配置文件关联的hint和与SQL计划基线关联的hint。然而，合并SQL配置文件与SQL计划基线有使用限制。实际上，就像下一部分介绍的那样，SQL计划基线的目的是强制使用特定的执行计划。结果，在考虑使用SQL计划基线之前，SQL配置文件的用处或许只是生成新的不被应用的执行计划。

11.7　SQL 计划管理

从11.1版本开始，SQL计划管理（SPM）取代了存储概要。其实，可以将SQL计划管理看作是存储概要的增强版。实际上，它们之间不仅具有相同的特性，并且SQL计划管理也具有同样的设计目的，即使执行环境或对象统计信息发生改变，也可以提供稳定的执行计划。此外，与存储概要一样，SQL计划管理也可以在不修改应用的情况下对应用进行优化。

警告　Oracle文档中唯一提到的SQL计划管理的用法是稳定执行计划。并未提及在不修改提交SQL语句的应用的情况下使用SQL计划管理来改变当前的执行计划（与某个给定的SQL语句相关），出于某些原因，我也选择忽略该功能。

下面是SQL计划管理包含的关键元素。

- SQL计划基线：用来稳定执行计划的实际对象。
- 语句日志：之前执行过的SQL语句列表。

- SQL管理基础（SMB）：存储SQL计划基线和语句日志的位置。需要的空间是在sysaux表空间中分配的。

11.7.1 工作原理

接下来的几个部分会介绍SQL计划管理是如何工作的。包括什么是SQL计划基线以及如何管理它们。要管理SQL计划基线，可以使用集成到企业管理器的图形界面。我们不会介绍这部分内容，因为在我看来，如果你懂得后台发生了什么，那么使用图形界面就不会有问题。

1. 什么是SQL计划基线

SQL计划基线是用来影响查询优化器生成执行计划的对象。更具体一点，SQL基线包含了一个或多个执行计划，而执行计划里包含一组hint。基本上，SQL计划基线用于强迫查询优化器针对给定SQL语句生成特定的执行计划。

警告 并不是所有的hint都会存储在SQL计划基线中。可以执行下面的查询来查明不会存储哪些hint：

```
SELECT name FROM v$sql_hint WHERE version_outline IS NULL
```

即使大多数hint无法保存在SQL计划基线中也不会影响到执行计划（例如，gather_plan_statistics），但其中一些会影响（例如，materialize和inline）。因此，对于有些执行计划，如果不增加hint，在SQL语句中是无法通过SQL计划基线来强制修改的。

SQL计划基线的其中一个优势它适用于某个特定SQL语句，并且不需要修改SQL语句本身。实际上，SQL计划基线存储在SQL基础管理平台上，并且查询优化器会自动选择它们。图11-8显示了选择期间执行的基本步骤。

(1) 首先，SQL语句按照正常方法解析。换句话说，查询优化器不使用SQL计划基线生成执行计划。

(2) 接着，查询优化器会将SQL语句标准化，使其不区分大小写并且与文本中的空格无关。计算出产生的SQL语句的签名，然后在SQL基础管理平台中执行查找。

如果找到相同签名的SQL计划基线，就会执行检查来确保SQL语句是最优的，并且与关联SQL计划基线的SQL语句是等价的。这一步是必要的检查，因为签名是散列值，因此可能存在冲突。

(3) 检查成功后，查询优化器会核实SQL计划基线是否包含没有使用SQL计划基线生成的执行计划。如果包含它并且接受（信任）它，就会执行它。

(4) 如果将另外一个接受的执行计划存储在SQL计划基线中，那么与它相关的hint会用来生成另一个执行计划。请注意如果SQL计划基线包含多个接受的执行计划，查询优化器会选择代价最小的那个。

(5) 最后，查询优化器检查利用SQL计划基线生成的执行计划是否会重现预估的执行计划。只有最后这个检查满足条件时，执行计划才可用。如果这个检查通不过，查询优化器会尝试其他接受的执行计划，如果所有的执行计划都无法重现，它会选择没有使用SQL计划基线生成的执行计划。

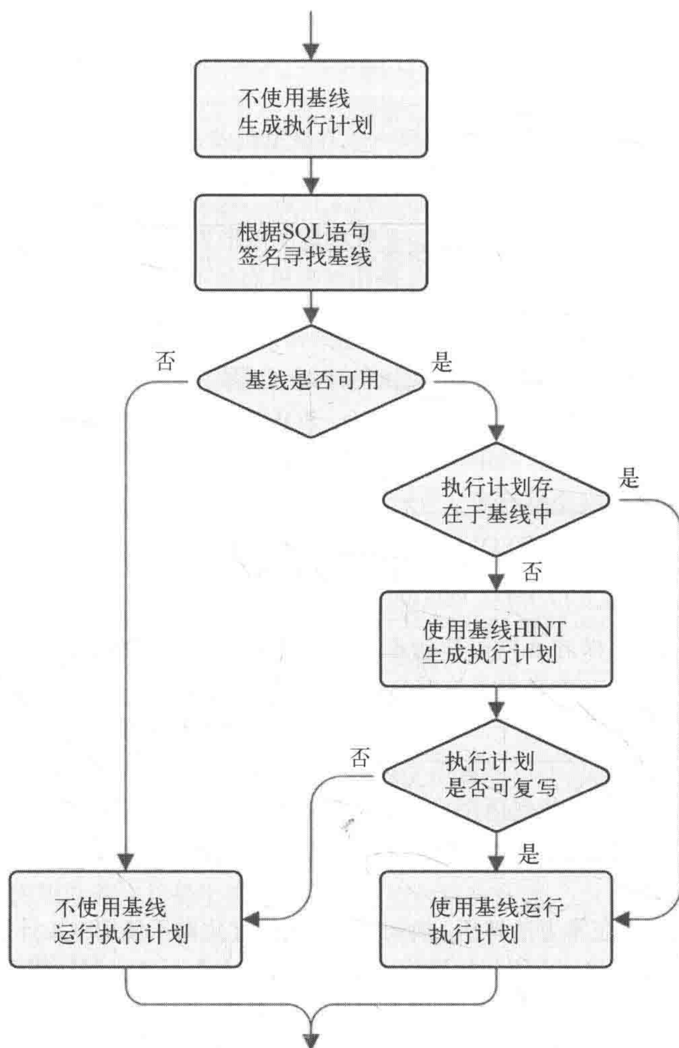


图11-8　SQL计划基线选择期间执行的主要步骤

2. 捕获SQL计划基线

可以通过几个步骤来捕获新的SQL计划基线。基本上，它们是由数据库引擎自动创建的，或由数据库管理员或开发人员手动创建。下面三部分分别介绍了三种方法。

- **自动捕获**

将初始化参数optimizer_capture_sql_plan_baselines设置为TRUE时，查询优化器会自动保存新的SQL计划基线。默认情况下，会将初始化参数设置为FALSE。可以在会话和系统级别更改它。

启用自动捕获时，查询优化会为每条多次执行（即至少执行两次）的SQL语句保存新的SQL计划基线。为此，它会在SQL基础管理平台中管理一个日志来插入每条它处理的SQL语句签名。这代表某

一SQL语句第一次执行后，它的签名仅会插入日志。然后，当同一个SQL语句第二次执行时，会创建仅包含当前执行计划的SQL计划基线并且标记为接受。从第三次执行开始，由于SQL计划基线已经与SQL语句相关联，因此查询优化器还会比较当前执行计划与SQL计划基线生成的执行计划。如果它们不匹配，这代表根据当前查询优化器的估算，最优的执行计划并不是存储在SQL计划基线中的那个。为了保存这个信息，会将当前执行计划添加到SQL计划基线中并且标记为不接受。然而，就像你之前看到的那样，当前执行计划无法使用。会强制查询优化器使用SQL计划基线生成的执行计划。图11-9总结了整个处理过程。

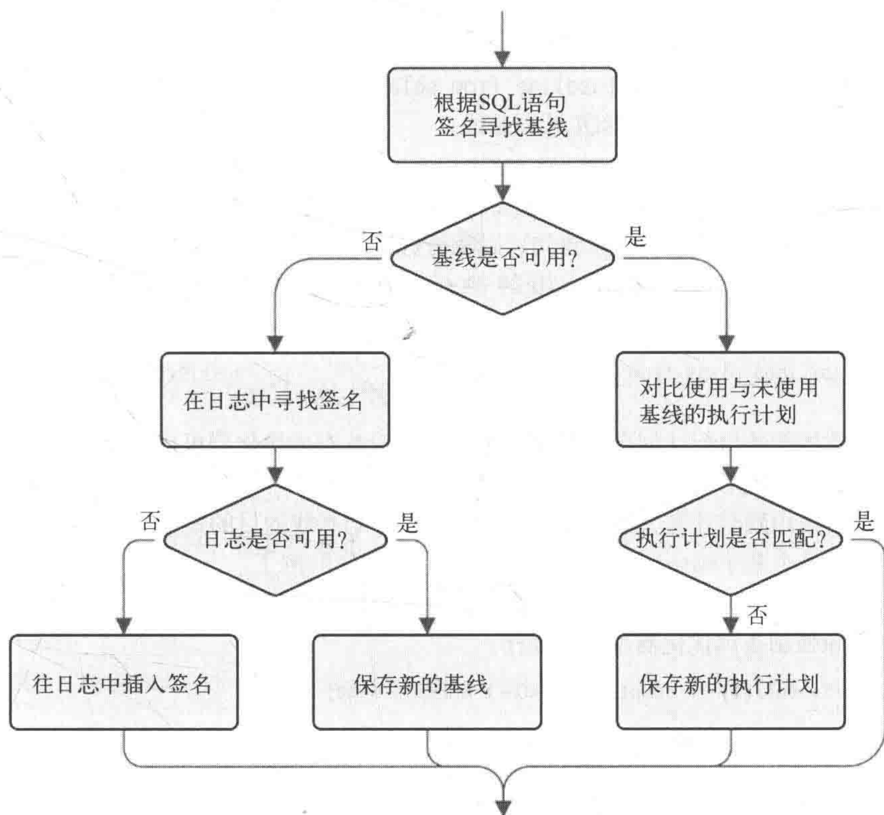


图11-9 自动捕获SQL计划基线期间执行的主要步骤

将某个新的执行计划存储到SQL计划基线中时，重点需要区分以下两种情况。

- 如果这是SQL计划基线的第一个执行计划，则会将执行计划存储为接受，因此，查询优化器将能够使用它。
- 如果这不是SQL计划基线的第一个执行计划，则会将它存储为不接受，因此，查询优化器无法使用它。“进化SQL计划基线”部分将介绍如何使SQL计划基线生效，以使其对查优化器可用。

- **从库缓存中加载**

要基于存储在库缓存中的游标手动将SQL计划基线加载进数据字典中，可以使用dbms_spm包下的load_plans_from_cursor_cache函数。

实际上，会多次重载函数来支持确定必须处理哪些游标的不同方法。这包含两种主要的可能。第一，通过指定以下属性之一来标识多个SQL语句。

- sql_text：SQL语句的文本。这个属性支持通配符（例如%）。
- parsing_schema_name：用来解析游标的模式名称。
- module：执行SQL语句的模块名称。
- action：执行SQL语句的动作名称。

举例说明，下面的调用引用自baseline_from_sqlarea1.sql脚本，为存储在库缓存中包含注释MySqlStm字符串的每个SQL语句创建SQL计划基线：

```
ret := dbms_spm.load_plans_from_cursor_cache(attribute_name  => 'sql_text',
                                             attribute_value => '%/* MySqlStm */%');
```

第二，通过它的SQL ID来标识SQL语句，以及可选执行计划的散列值。如果散列值没有指定或设置为NULL，所有对指定SQL语句可用的执行计划都会被加载。下面的调用，引用自baseline_from_sqlarea2.sql脚本：

```
ret := dbms_spm.load_plans_from_cursor_cache(sql_id          => '2y5r75r8y3sj0',
                                             plan_hash_value => NULL);
```

使用这些函数加载的执行计划会被存储为可接受，因此查询优化器可以立即利用它们。

在之前的例子中，SQL计划基线基于库缓存中找到的SQL语句的文本。这只有在你想确保当前的执行计划未来也会被用到时才有关系。有时，使用SQL计划基线的目的是优化SQL语句而不用修改应用。让我们看这样一个基于baseline_from_sqlarea3.sql脚本的例子。

假设应用执行以下SQL语句。查询优化器基于全表扫描生成执行计划。这是因为在SQL语句中包含一个hint，该hint强制查询优化器指向此操作：

```
SQL> SELECT /*+ full(t) */ count(pad) FROM t WHERE n = 42;

SQL> SELECT * FROM table(dbms_xplan.display_cursor);

-----------------------------------
| Id  | Operation          | Name |
-----------------------------------
|   0 | SELECT STATEMENT   |      |
|   1 |  SORT AGGREGATE    |      |
|*  2 |   TABLE ACCESS FULL| T    |
-----------------------------------

  2 - filter("N"=42)
```

你会注意到限制列（*n*）上有索引存在。接着你或许想知道当使用索引时性能会怎样。因此，正如下面的例子所示，使用某个hint来执行SQL语句，以确保能够使用索引：

```
SQL> SELECT /*+ index(t) */ count(pad) FROM t WHERE n = 42;
```

```
SQL> SELECT * FROM table(dbms_xplan.display_cursor);

SQL_ID dat4n4845zdxc, child number 0
------------------------------------

Plan hash value: 3694077449

---------------------------------------------
| Id  | Operation                    | Name |
---------------------------------------------
|   0 | SELECT STATEMENT             |      |
|   1 |  SORT AGGREGATE              |      |
|   2 |   TABLE ACCESS BY INDEX ROWID| T    |
|*  3 |    INDEX RANGE SCAN          | I    |
---------------------------------------------

   3 - access("N"=42)
```

如果第二个执行计划比第一个更有效率，你的目的就是让应用使用它。如果无法更改应用来删除或者修改hint，可以利用SQL计划基线来解决这个问题。可以自动或者手动创建SQL计划基线来达到目的。在这种情况下，你决定使用初始化参数optimizer_capture_sql_plan_baselines：

```
SQL> ALTER SESSION SET optimizer_capture_sql_plan_baselines = TRUE;

SQL> SELECT /*+ full(t) */ count(pad) FROM t WHERE n = 42;

SQL> SELECT /*+ full(t) */ count(pad) FROM t WHERE n = 42;

SQL> ALTER SESSION SET optimizer_capture_sql_plan_baselines = FALSE;
```

一旦SQL计划基线创建好，就要检查是否真的使用了它。通过dbms_xplan包可以看到SQL计划名，用来标识生成执行计划的SQL计划基线：

```
SQL> SELECT /*+ full(t) */ count(pad) FROM t WHERE n = 42;

SQL> SELECT * FROM table(dbms_xplan.display_cursor);

-----------------------------------
| Id  | Operation          | Name |
-----------------------------------
|   0 | SELECT STATEMENT   |      |
|   1 |  SORT AGGREGATE    |      |
|*  2 |   TABLE ACCESS FULL| T    |
-----------------------------------

  2 - filter("N"=42)

Note
-----
   - SQL plan baseline SQL_PLAN_3u6sbgq7v4u8z3fdbb376 used for this statement
```

接着，基于之前的输出提供的SQL计划名，通过dba_sql_plan_baselines视图，可以找到SQL计划基线的标识符，即SQL句柄（SQL handle）：

```
SQL> SELECT sql_handle
```

```
  2  FROM dba_sql_plan_baselines
  3  WHERE plan_name = 'SQL_PLAN_3u6sbgq7v4u8z3fdbb376';

SQL_HANDLE
--------------------
SQL_3d1b0b7d8fb2691f
```

最后，你使用SQL计划基线替换执行计划。要这么做，需要加载执行索引扫描的执行计划，并移除执行全表扫描的执行计划。前者被SQL标识符以及执行计划散列值引用，后者被SQL句柄和SQL计划名引用：

```
ret := dbms_spm.load_plans_from_cursor_cache(sql_handle      => 'SQL_3d1b0b7d8fb2691f',
                                             sql_id          => 'dat4n4845zdxc',
                                             plan_hash_value => '3694077449');

ret := dbms_spm.drop_sql_plan_baseline(sql_handle => 'SQL_3d1b0b7d8fb2691f',
                                       plan_name => 'SQL_PLAN_3u6sbgq7v4u8z3fdbb376');
```

要检查替换是否成功，可以测试新的SQL计划基线。请注意即使SQL语句包含full hint，执行计划也不会再使用全表扫描。

注意　在实践中，不恰当的hint经常导致低效的执行计划。能够使用这部分技术来覆盖它们是非常有用的。

```
SQL> SELECT /*+ full(t) */ count(pad) FROM t WHERE n = 42;

SQL> SELECT * FROM table(dbms_xplan.display_cursor);

--------------------------------------------------
| Id  | Operation                      | Name |
--------------------------------------------------
|   0 | SELECT STATEMENT               |      |
|   1 |  SORT AGGREGATE                |      |
|   2 |   TABLE ACCESS BY INDEX ROWID| T    |
|*  3 |    INDEX RANGE SCAN            | I    |
--------------------------------------------------

   3 - access("N"=42)

Note
-----
   - SQL plan baseline SQL_PLAN_3u6sbgq7v4u8z59340d78 used for this statement
```

要想知道SQL计划基线是否被用于某条SQL语句，也可以查询v$sql视图的sql_plan_baseline列。请注意该列显示的是SQL计划名，不是dbms_xplan包显示的SQL句柄。

● **从SQL调优集中加载**

dbms_spm包下的load_plans_from_sqlset函数，可以从SQL调优集中加载SQL计划基线。加载仅需要指定所有者（owner）和SQL调优集名称。下面的调用，节选自baseline_from_sqlset.sql脚本：

```
ret := dbms_spm.load_plans_from_sqlset(sqlset_name => 'test_sqlset',
                                       sqlset_owner => user);
```

使用该函数加载的执行计划会被存储为可接受。因此，查询优化器可以立即利用它们。

升级到新版本可以用到该函数。实际上，10.2版本的数据库创建的SQL调优集也可以加载到11.2版本的数据库中。Baseline_upgrade_10g.sql和baseline_upgrade_11g.sql脚本列举了这样的应用。

3. 显示SQL计划基线

通过dba_sql_plan_baseline视图（从12.1版本之后，也可以查询cdb_sql_plan_baselines视图）可以显示可用SQL计划基线的基本信息。要显示详细信息，可以使用dbms_xplan包下的display_sql_plan_baseline函数。请注意它与第10章讨论的dbms_xplan包下的另一个函数相似。下面的例子展示了它可以显示的信息：

```
SQL> SELECT *
  2  FROM table(dbms_xplan.display_sql_plan_baseline(sql_handle => 'SQL_971650b23f790eb7'));

--------------------------------------------------------------------------------
SQL handle: SQL_971650b23f790eb7
SQL text: SELECT /* MySqlStm */ count(pad) FROM t WHERE n = 28
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
Plan name: SQL_PLAN_9f5khq8zrk3pr3fdbb376         Plan id: 1071362934
Enabled: YES     Fixed: NO      Accepted: YES     Origin : MANUAL-LOAD
--------------------------------------------------------------------------------

Plan hash value: 2966233522

---------------------------------------------------------------------------
| Id  | Operation           | Name | Rows | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |    1 |   505 |    20   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE     |      |    1 |   505 |            |          |
|*  2 |   TABLE ACCESS FULL| T    |    1 |   505 |    20   (0)| 00:00:01 |
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

  2 - filter("N"=28)
```

警告　要想在11.1版本和11.2版本中正确显示SQL计划基线的信息，display_sql_plan_baseline函数必须能够重现与其关联的执行计划。如果函数无法实现，它会返回错误的结果甚至报错信息。为了避免这样的问题，从12.1版本开始，执行计划为了报告而存储在SQL基础管理平台中。可以执行baseline_unreproducible.sql脚本来观察输出结果以免存在不可重现的执行计划。

不幸的是，必须在11.1版本中查询数据字典来显示与SQL计划基线关联的hint列表。下面的SQL语句显示了一个示例。请注意，由于hint被存储成XML格式，因此需要转换成可读的输出：

```
SQL> SELECT extractValue(value(h),'.') AS hint
  2  FROM sys.sqlobj$data od, sys.sqlobj$ so,
  3       table(xmlsequence(extract(xmltype(od.comp_data),'/outline_data/hint'))) h
  4  WHERE so.name = 'SQL_PLAN_9f5khq8zrk3pr3fdbb376'
  5  AND so.signature = od.signature
  6  AND so.category = od.category
  7  AND so.obj_type = od.obj_type
  8  AND so.plan_id = od.plan_id;

HINT
------------------------------------
IGNORE_OPTIM_EMBEDDED_HINTS
OPTIMIZER_FEATURES_ENABLE('11.2.0.3')
DB_VERSION('11.2.0.3')
ALL_ROWS
OUTLINE_LEAF(@"SEL$1")
FULL(@"SEL$1" "T"@"SEL$1")
```

然而，从11.2版本之后，也同样可以使用display_sql_plan_baseline函数来显示hint列表。实际上，对于dbms_xplan包下的其他函数，format参数都可以用来影响它们的输出。下面的例子引用自将format参数设置为outline时产生的输出：

```
SQL> SELECT *
  2  FROM table(dbms_xplan.display_sql_plan_baseline(sql_handle => 'SQL_971650b23f790eb7',
  3                                                  format     => 'outline'));

Outline Data from SMB:

  /*+
    BEGIN_OUTLINE_DATA
    IGNORE_OPTIM_EMBEDDED_HINTS
    OPTIMIZER_FEATURES_ENABLE('11.2.0.3')
    DB_VERSION('11.2.0.3')
    ALL_ROWS
    OUTLINE_LEAF(@"SEL$1")
    FULL(@"SEL$1" "T"@"SEL$1")
    END_OUTLINE_DATA
*/
```

4. 进化SQL计划基线

如果查询优化器生成的执行计划不是与它正在优化的SQL语句相关联的SQL计划基线中的现有执行计划，就会将一个新的未接受的执行计划自动添加到SQL计划基线中。即使查询优化器无法立即使用未接受的执行计划时也会发生。这样做的目的是保留存在另一个可能更好的执行计划的信息。要验证一个未接受的执行计划是否要比SQL计划基线生成的执行计划更好时，就必须尝试进化(evolution)。这仅仅是要求SQL引擎用不同的执行计划执行SQL语句，并查明未接受的SQL计划基线的性能是否比接受的SQL计划基线的性能更好。如果答案是肯定的，则会将未接受的SQL计划基线设置为接受。

警告 在进化期间SQL引擎使用特殊的方式处理SQL语句。实际上，对于INSERT/UPDATE/MERGE/DELETE语句，只是访问数据而不会修改数据。因此，SQL语句仅是部分执行。然而，我认为这没什么问题。实际上，修改数据的操作总是会执行相同的工作，而并不取决于如何访问修改的数据。

可以使用dbms_spm包下的evolve_sql_plan_baseline函数来执行进化。要调用这个函数，除了使用sql_handle和/或plan_name参数来确定SQL计划基线外，还需要以下参数。

- time_limit：以分钟为单位，进化可持续的时间。这个参数接受自然数或dbms_spm.auto_limit和dbms_spm.no_limit常量。
- Verify：如果设置为yes（默认），则会执行SQL语句以验证性能。如果设置为no，就不会执行验证，并且SQL计划基线也会简单地变为接受。
- Commit：如果设置为yes（默认），数据字典会根据进化的结果做修改。如果设置为no，并且verify参数设置为yes，则会执行验证，但不修改数据字典。

报告是函数的返回值，它提供了进化的详细信息。下面的例子，引用自baseline_automatic.sql脚本生成的输出，显示SQL语句用来启动进化，并且结果报告指出SQL计划基线被进化了（包括导致这个决定的统计信息）：

```
SQL> SELECT dbms_spm.evolve_sql_plan_baseline(sql_handle => 'SQL_492bdb47e8861a89',
  2                                           plan_name => '',
  3                                           time_limit => 10,
  4                                           verify => 'yes',
  5                                           commit => 'yes')
  6  FROM dual;

-------------------------------------------------------------------------------
                        Evolve SQL Plan Baseline Report
-------------------------------------------------------------------------------

Inputs:
-------
  SQL_HANDLE = SQL_492bdb47e8861a89
  PLAN_NAME  =
  TIME_LIMIT = 10
  VERIFY     = yes
  COMMIT     = yes

Plan: SQL_PLAN_4kayv8zn8c6n959340d78
------------------------------------
  Plan was verified: Time used .05 seconds.
  Plan passed performance criterion: 24.59 times better than baseline plan.
  Plan was changed to an accepted plan.

                            Baseline Plan      Test Plan       Stats Ratio
                            -------------      ---------       -----------
Execution Status:                COMPLETE       COMPLETE
Rows Processed:                         1              1
Elapsed Time(ms):                    .527           .054              9.76
CPU Time(ms):                        .333           .111                 3
Buffer Gets:                           74              3             24.67
Physical Read Requests:                 0              0
Physical Write Requests:                0              0
Physical Read Bytes:                    0              0
Physical Write Bytes:                   0              0
Executions:                             1              1
```

```
----------------------------------------------------------------------------
                             Report Summary
----------------------------------------------------------------------------
Number of plans verified: 1
Number of plans accepted: 1
```

除了刚刚介绍的手动进化外，SQL计划基线的自动进化需要Tuning Pack选件支持。原因是，在维护窗口期间，SQL优化顾问处理SQL语句会对系统造成重大影响。在可能的情况下，优化顾问提供建议来改进它们的响应时间。如果优化顾问注意到未接受的SQL计划基线比接受的SQL计划基线的性能更好，它会建议SQL配置文件使用接受的SQL计划基线。显然，如果接受SQL配置文件，则也会接受SQL计划基线。因此，只要SQL计划基线自动接受，那么SQL优化顾问生成的SQL配置文件也会自动接受。

必须要指出的是SQL配置文件只有在针对SQL优化顾问的accept_sql_profile参数设置为TRUE时才会自动接受。默认情况下是FALSE。你可以借助类似以下查询通过dba_advisor_parameters视图检查它的值（请注意，同样user和12.1及之后版本中与cdb相关的视图也存在）：

```
SQL> SELECT parameter_value
  2  FROM dba_advisor_parameters
  3  WHERE task_name = 'SYS_AUTO_SQL_TUNING_TASK'
  4  AND parameter_name = 'ACCEPT_SQL_PROFILES';

PARAMETER_VALUE
---------------
FALSE
```

dbms_auto_sqltune包提供了set_auto_tuning_task_parameter过程用来更改accept_sql_profiles参数的值。下面的例子展示如何将参数设置为TRUE来激活SQL配置文件的自动接受：

```
dbms_auto_sqltune.set_auto_tuning_task_parameter(parameter => 'ACCEPT_SQL_PROFILES',
                                                 value     => 'TRUE');
```

从12.1版本开始，又有了一个新的顾问叫作SPM进化顾问（SPM Evolve Advisor）。它的目的是为与SQL计划基线相关联的未接受执行计划执行进化。它在维护窗口期间执行，这一点与其他顾问一样。可以使用dbms_spm包下的report_auto_evolve_task函数来显示SPM进化顾问都做了什么。如果只调用这个函数而不加任何参数，它会显示最后一次执行的报告。下面的例子展示了当最后三次执行发生后，如何通过dba_advisor_executions视图找到它（请注意，同样user和12.1及之后版本中与cdb相关的视图也存在），以及如何显示某个执行的报告：

```
SQL> SELECT *
  2  FROM (
  3    SELECT execution_name, execution_start
  4    FROM dba_advisor_executions
  5    WHERE task_name = 'SYS_AUTO_SPM_EVOLVE_TASK'
  6    ORDER BY execution_start DESC
  7  )
  8  WHERE rownum <= 3;

EXECUTION_NAME EXECUTION_START
-------------- ----------------
EXEC_6294      23-APR-14
```

```
EXEC_6182       22-APR-14
EXEC_6082       21-APR-14

SQL> SELECT dbms_spm.report_auto_evolve_task(execution_name => 'EXEC_6294')
  2  FROM dual;

GENERAL INFORMATION SECTION
---------------------------------------------------------

Task Information:
---------------------------------------------
Task Name            : SYS_AUTO_SPM_EVOLVE_TASK
Task Owner           : SYS
Description          : Automatic SPM Evolve Task
Execution Name       : EXEC_6294
Execution Type       : SPM EVOLVE
Scope                : COMPREHENSIVE
Status               : COMPLETED
Started              : 04/23/2014 22:00:19
Finished             : 04/23/2014 22:00:19
Last Updated         : 04/23/2014 22:00:19
Global Time Limit    : 3600
Per-Plan Time Limit : UNUSED
Number of Errors     : 0
---------------------------------------------------------

SUMMARY SECTION
---------------------------------------------------------
  Number of plans processed : 0
  Number of findings        : 0
  Number of recommendations : 0
  Number of errors          : 0
---------------------------------------------------------
```

5. 修改SQL计划基线

创建SQL计划基线时，可以使用dbms_spm包下的alter_sql_plan_baseline过程来修改某些指定的参数。Sql_handle和plan_name参数确定被修改的SQL计划基线。必须指定这两个参数中的一个。Attribute_name和attribute_value参数确定被修改的属性以及它们的新值。Attribute_name参数可以接受以下值。

- enabled：可以将这个属性设置为yes或no，但只有在设置为yes时，查询优化器才可以使用SQL计划基线。
- fixed：将这个属性设置为yes时，不会将新的执行计划添加到SQL计划基线中，结果就是之后它都不能进化。此外，如果SQL计划基线包含多个可接受的执行计划，固定的执行计划要比未固定的好。可以将这个值设置为yes或no。
- autopurge：这个属性设置为yes的SQL计划基线会在一段时间不使用后自动删除（保留时间的配置会在稍后的“删除SQL计划基线“部分介绍）。可以将这个值设置为yes或no。
- plan_name：这个属性用来更改SQL计划名。它可以是不超过30个字符的任意字符串。
- description：这个属性用来为SQL计划基线附加描述。它可以是不超过500个字符的任意字符串。

下面的调用中禁用与执行计划关联的SQL计划基线：

```
ret := dbms_spm.alter_sql_plan_baseline(sql_handle      => 'SQL_492bdb47e8861a89',
                                        plan_name       => 'SQL_PLAN_4kayv8zn8c6n93fdbb376',
                                        attribute_name  => 'enabled',
                                        attribute_value => 'no');
```

6. 激活SQL计划基线

查询优化器只有在初始化参数optimizer_use_sql_plan_baselines设置为TRUE（这是默认值）时才会使用SQL计划基线。可以在会话或系统级别更改它。

7. 移动SQL计划基线

dbms_spm包提供了多个过程用来在数据库之间移动SQL计划基线。比如，当SQL计划基线需要在开发环境或测试数据库中生成然后移动到生产环境中时。正如图11-10所示，会提供以下特性。

- 可以使用create_stgtab_baseline过程来创建临时表。
- 可以使用pack_stgtab_baseline函数将SQL计划基线从数据字典复制到临时表中。
- 可以使用unpack_stgtab_baseline函数将SQL计划基线从临时表复制到数据字典中。

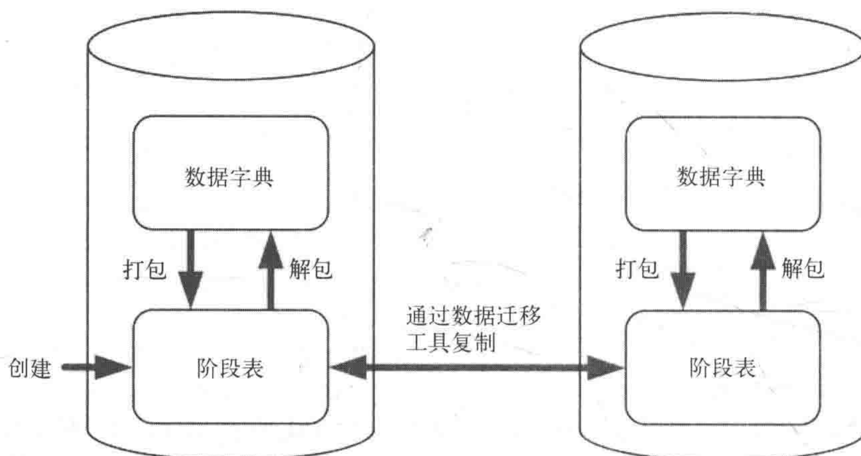


图11-10　使用dbms_spm包移动SQL计划基线

请注意，在数据库之间移动临时表依靠的是数据移动技术（例如，数据泵（Data Pump或旧有的导出（export）和导入（import）程序），而不是依靠dbms_spm包本身（参见图11-10）。

下面的例子引用自baseline_clone.sql脚本，展示了如何将SQL计划基线从一个数据库复制到另一个。首先，在当前模式下创建mystgtab临时表：

```
dbms_spm.create_stgtab_baseline(table_name      => 'MYSTGTAB',
                                table_owner     => user,
                                tablespace_name => 'USERS');
```

接着将SQL计划基线从数据字典复制到临时表中。可以通过以下四种方法识别要处理哪些SQL计划基线。

- 通过sql_handle和可选的plan_name参数来准确识别SQL计划基线。

- ❑ 选择所有在SQL语句文本中包含特定字符串的SQL计划基线。为此，可以使用支持通配符（例如，%）的sql_text参数。请注意该参数区分大小写。
- ❑ 选择所有符合以下一个或多个参数的SQL计划基线：creator、origin、enabled、accepted、fixed、module和action。如果指定了多个参数，那么就需要满足它们的所有值。
- ❑ 处理所有SQL计划基线。这种方法不需要指定参数。

下面的调用展示了如何准确识别SQL计划基线：

```
ret := dbms_spm.pack_stgtab_baseline(table_name  => 'MYSTGTAB',
                                     table_owner => user,
                                     sql_handle  => 'SQL_492bdb47e8861a89',
                                     plan_name   => 'SQL_PLAN_4kayv8zn8c6n93fdbb376');
```

此时，依靠数据移动程序，将mystgtab临时表从一个数据库复制到另一个。

最后，将SQL计划基线从临时表复制到目标数据库的数据字典中。要识别处理的SQL计划基线，可使用与pack_stgtab_baseline函数同样的方法。下面的调用展示了通过SQL语句的文本来识别SQL计划基线：

```
ret := dbms_spm.unpack_stgtab_baseline(table_name  => 'MYSTGTAB',
                                       table_owner => user,
                                       sql_text    => '%FROM t%');
```

8. 删除SQL计划基线

可以使用dbms_spm包下的drop_sql_plan_baseline过程从数据字典中删除SQL计划基线。sql_handle和sql_name参数指定要删除的执行计划和/或SQL计划基线。这两个参数至少需要设置一个。下面的调用说明了这一点：

```
ret := dbms_spm.drop_sql_plan_baseline(sql_handle => 'SQL_492bdb47e8861a89',
                                       plan_name  => 'SQL_PLAN_4kayv8zn8c6n93fdbb376');
```

未使用的SQL计划基线有自动删除条件设置的属性设置为yes，在一段时间后自动删除。默认的周期是53周。当前值可以使用类似以下的查询通过dba_sql_management_config视图查看（在12.1及之后版本中，也存在cdb版本的视图）：

```
SQL> SELECT parameter_value
  2  FROM dba_sql_management_config
  3  WHERE parameter_name = 'PLAN_RETENTION_WEEKS';

PARAMETER_VALUE
---------------
             53
```

可以调用dbms_spm包下的configure过程来修改保留期。可以更改为5至523周。下面的例子展示了如何更改为12周。如果将parameter_value参数设置为NULL，就会恢复成默认值：

```
dbms_spm.configure(parameter_name  => 'plan_retention_weeks',
                   parameter_value => 12);
```

9. 权限

自动捕获SQL计划基线时（即，通过将初始化参数optimizer_capture_sql_plan_baselines设置为

TRUE来实现)，并不需要特别的权限来创建它们。

dbms_spm包只能由拥有administer sql management object系统权限的用户执行(默认情况下，dba角色拥有该权限)。SQL计划基线并不存在对象权限。

最终用户不需要特定权限也可以使用SQL计划基线。

11.7.2　何时使用

在两种情况下，需要考虑使用SQL计划基线。第一，需要优化一条SQL语句而不能在应用中修改它时(例如，无法增加hint)。第二，遇到任何原因导致的执行计划不稳定时。由于SQL计划基线的目的是强制查询优化器为给定SQL语句选择指定执行计划，因此仅当需要明确限制查询优化器选择单个执行计划时才会使用该技巧。

遗憾的是，SQL计划基线仅可以在企业版中使用。标准版请使用存储概要替代。

11.7.3　陷阱和谬误

SQL计划基线最重要的属性之一是它们是从代码中分离的。然而这也会带来问题。实际上，由于在SQL计划基线与SQL语句之间并没有直接的关联，开发人员很可能会彻底忽略SQL计划基线的存在。结果，如果开发人员修改SQL语句将会导致它的签名发生改变，这样SQL计划基线就不会在生效了。同样，当你部署一个应用需要依靠SQL计划基线来保证执行正确时，必须记得在数据库设置期间安装它们。

需要注意的是，SQL计划基线依赖的对象删除时它并不会被删除。但这并不是问题。例如，如果一个表或索引因为它必须重组或移动而需要重建，那么SQL计划基线没被删除就是好事。否则，就有必要重建它们。总之，未使用的SQL计划基线会在周期过后被删除。

两个有相同文本的SQL语句拥有相同的签名。即使它们引用的对象在不同的模式下。这代表单个SQL计划基线可以被两个同名但是不同模式的表使用。再次强调，你需要小心，尤其是数据库里同样的对象有多个副本时。

SQL计划基线不支持引用远程数据库表的SQL语句。

在11.2.0.2及之前的版本中，因为Oracle Support文档*SQL profile not used in the Acive Physical Standby* (10050057.8)中描述的bug，导致SQL计划基线在Active Data Guard环境下受限。可以在主实例上使用SQL计划基线，但并不总是能在备用实例上使用。

SQL计划基线存储在sysaux表空间中的SQL基础管理平台上。默认情况下，该表空间最大10%的空间会留给它们。可以通过dba_sql_management_config视图显示当前值：

```
SQL> SELECT parameter_value
  2  FROM dba_sql_management_config
  3  WHERE parameter_name = 'SPACE_BUDGET_PERCENT';

PARAMETER_VALUE
---------------
             10
```

当超过限制的时候，会将警告信息写入alert日志中。要改变默认的限制，可以使用dbms_spm包下

的config过程。值可以填写1% ~ 50%。下面的例子展示了如何将它的值更改为5%。如果将parameter_value参数设置为NULL，参数就会恢复默认值：

```
dbms_spm.configure(parameter_name  => 'space_budget_percent',
                   parameter_value => 5);
```

当SQL语句有SQL配置文件和存储概要时，查询优化器会仅使用存储概要。当然，前提是存储概要处于激活状态。

当SQL语句有SQL配置文件和SQL计划基线时，查询优化器会尝试合并与SQL配置文件关联的hint和与SQL计划基线关联的hint。然而，合并SQL配置文件与SQL计划基线有使用限制。实际上，SQL计划基线的目的是强制使用特定的执行计划。结果，在考虑使用SQL计划基线之前，SQL配置文件的用处或许只是生成新的不被应用的执行计划。

11.8 小结

本章描述了多种SQL优化技巧。选择其中一个并不总是那么简单。不过，如果你理解它们的工作原理和使用它们的利弊，那么选择起来就容易多了。即便如此，实际中不同的场景也会限制你使用不同的技巧。这或许是因为技巧的限制或授权问题。

第12章专门介绍解析，这是执行SQL语句的核心步骤之一。解析之所以重要是因为，当查询优化器生成执行计划时，为了总是能有高效地执行计划，你会希望解析每条由数据库引擎执行的SQL语句。但是相反，解析也是一个昂贵的操作。结果就是必须将它降到最低限度，并且执行计划应该尽可能地被重用，但不是重用太多。这表示执行计划并不总是高效的。为了最大可能地利用数据库引擎，必须理解工作原理和不同特性的利弊。

第 12 章

解析

12

解析对全部性能影响的可变因素非常多。在某些情况下，可以简单地忽略它。在其他情况下，它是造成性能问题的主要原因。如果存在解析问题，这通常代表应用不能正确处理它。这是个主要问题，因为通常要改变应用的行为，你需要修改相应的代码。开发人员需要知道解析的影响以及如何在写代码时尽可能避免相关问题。

第2章介绍了游标的生命周期和解析的工作原理。本章介绍如何识别、解决和避开解析问题。我也会介绍与解析有关的总开销。最后，我会介绍用来减少解析活动的通用应用编程接口提供的特性。

12.1 识别解析问题

当寻找解析问题时，很容易会遇到强迫性的混乱优化。发生这类问题的原因是，多个动态性能视图包含的计数器详细记录了软解析、硬解析和执行的次数。这些计数器和基于它们的比率一样，都是没用的，因为它们没有提供关于解析花费时间的信息。请注意对于解析，这才是真正的问题，因为它们没有标准周期。实际上，根据SQL语句的复杂度和它引用的对象，解析的周期通常会相差几个数量级。简单地说，这些计数器只能告诉你数据库引擎是否完成少量或大量的解析，而没有关于是否存在问题的信息。因此，实际中它们只用来做趋势分析。

如果你遵循第一部分和第二部分提供的建议，那么应该清晰地知道，唯一有效识别解析问题的方法，就是衡量数据库引擎花费了多少时间来解析SQL语句。如果要查找单个会话或是整个系统的全部时间信息，可以查询提供时间模型统计信息的动态性能视图。这些视图包括v$sess_time_model、v$sys_time_model和在12.1多租户环境下的v$con_sys_time_model。

例如，下面查询的输出显示了一个会话花费了大量时间（将近59%）来解析SQL语句的信息：

```
SQL> WITH
  2    db_time AS (SELECT sid, value
  3                FROM v$sess_time_model
  4                WHERE sid = 137
  5                AND stat_name = 'DB time')
  6  SELECT ses.stat_name AS statistic,
  7         round(ses.value / 1E6, 3) AS seconds,
  8         round(ses.value / nullif(tot.value, 0) * 1E2, 1) AS "%"
  9  FROM v$sess_time_model ses, db_time tot
 10  WHERE ses.sid = tot.sid
 11  AND ses.stat_name <> 'DB time'
 12  AND ses.value > 0
```

```
 13  ORDER BY ses.value DESC;

STATISTIC                                   SECONDS          %
---------------------------------------- ---------- ----------
DB CPU                                       18.204       99.3
parse time elapsed                           10.749       58.6
hard parse elapsed time                       8.048       43.9
sql execute elapsed time                      1.968       10.7
connection management call elapsed time        .021         .1
PL/SQL execution elapsed time                  .009         .1
repeated bind elapsed time                        0          0
```

类似于这个查询所提供的信息可以用来判断解析是否存在问题。不幸的是，通过动态性能视图提供的时间模型统计信息，并不能帮助找出是哪个SQL语句导致的问题！

如果要寻找的是证据而不是线索，那么只有两个信息来源可以使用：由SQL跟踪生成的输出，和来自v$active_session_history或dba_hist_active_sess_history的活动会话历史。实际上，在SQL语句级别上，这些是仅有的可以提供关于解析定时信息的来源。这就是为什么本章我会仅基于SQL跟踪和活动会话历史来讨论解析问题的识别。

注意 如果想使用活动会话历史来分析解析问题，需要知道四个限制。第一，使用活动会话历史不仅需要企业版，还需要Diagnostics Pack选件。第二，仅在11.1及之后版本中，活动会话历史才提供用于分析解析问题（in_parse和in_hard_parse标识）的必要信息。第三，不能使用企业管理器来做分析。第四，既然SQL语句的文本无法直接通过活动会话历史获得（只能得到SQL ID），那么获得的信息并不一定足够用来识别导致解析问题的SQL语句。

解析问题主要有两种。第一种与持续时间非常短的解析有关，称为快速解析（quick parse）。当然需要大量执行才会引起注意。第二种解析问题与持续时间很长的解析有关，称为长解析（long parses）。通常是在SQL语句相当复杂或查询优化器需要很长时间才能生成高效执行计划时才会出现。这种情况下与执行次数无关。

在接下来的两节里，我会介绍用来识别这两类解析问题的方法。既然对于这两种问题的识别没有本质的区别，我仅会全面介绍第一种。

12.1.1 快速解析

接下来介绍如何定位快速解析导致的性能问题。针对11.2.0.3版本的数据库，执行ParsingTest1.java文件中的类来生成负载样例。同样在PL/SQL、C(OCI)、C#（ODP.NET）和PHP（PECL OCI8扩展）中也实现了同样的处理过程。鉴于在第3章中介绍过两个探查器，TKPROF和TVD$XTAT，我会针对这两个探查器的输出文件来讨论相同的例子。可以在ParsingTest1.zip文件中找到跟踪文件和输出文件。

1. 使用TKPROF

正如第3章中建议的那样，TKPROF使用以下选项执行：

```
tkprof <trace file> <output file> sys=no sort=prsela,exeela,fchela
```

要开始分析输出文件，最好先看一下最后几行。在本例里，需要重点注意的是处理持续了大约14秒，应用程序执行了10 000个SQL语句，并且所有SQL语句都不相同（user SQL statements与unique SQL statements相等）。

```
     1  session in tracefile.
 10000  user SQL statements in trace file.
     0  internal SQL statements in trace file.
 10000  SQL statements in trace file.
 10000  unique SQL statements in trace file.
120060  lines in trace file.
    14  elapsed seconds in trace file.
```

接着，需要检查输出中的第一个SQL语句执行了多长时间。由于指定了sort选项，SQL语句可以根据其响应时间进行排序。有趣的是，第一个游标的响应时间要小于百分之一秒（0.00）。换句话说，所有SQL语句的执行都低于百分之一秒。实际上，平均一个执行持续了1.4毫秒（14/10 000）。这很明显意味着是短时间内处理的大量SQL语句占用了大量的响应时间，而不是少量长时间运行的SQL语句。

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ---------- ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.00       0.00          0          2          0           0
------- ------  -------- ---------- ---------- ---------- ---------- ----------
total        3      0.00       0.00          0          2          0           0
```

这种情况下，要想判断是不是解析的问题，就必须检查总计的部分。根据执行统计信息，解析时间大约占用整个执行时间的95%（5.7/6）。这明显证明了数据库引擎除了解析什么都没做。

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ---------- ----------
Parse    10000      5.54       5.70          0          0          0           0
Execute  10000      0.17       0.15          0          0          0           0
Fetch    10000      0.13       0.14          0      23051          0        3048
------- ------  -------- ---------- ---------- ---------- ---------- ----------
total    30000      5.86       6.00          0      23051          0        3048
```

下面这行也显示这10 000个解析都是硬解析。注意，即使高比例的硬解析通常并不是我们想要的，但这未必就有问题。但这证明了存在次优的部分。

```
Misses in library cache during parse: 10000
```

执行统计信息的问题是缺少大约57%（1–6.00 / 14）的响应时间。实际上，通过查看汇总等待事件的表，可以看到等待客户端用去了6.24秒。然而，仍然有大约2秒（14 – 6.00 – 6.24）下落不明。

```
Event waited on                             Times   Max. Wait  Total Waited
----------------------------------------   Waited  ----------  ------------
SQL*Net message to client                   10000        0.00          0.02
SQL*Net message from client                 10000        0.02          6.24
latch: shared pool                              5        0.00          0.00
log file sync                                   1        0.00          0.00
```

当你知道解析出了问题时，明智的做法是查看一下SQL语句。本例中，查看它们其中的一些即可（下面是排名前五位的SQL语句），很明显它们都非常类似。只有在WHERE子句中用到的文字不同。这是不用绑定变量的典型案例。

```
SELECT pad FROM t WHERE val = 0
SELECT pad FROM t WHERE val = 2139
SELECT pad FROM t WHERE val = 9035
SELECT pad FROM t WHERE val = 8488
SELECT pad FROM t WHERE val = 1
```

这种情况的问题是，TKPROF无法识别只有文字不同的SQL语句。实际上，即使当aggregate选项设置为yes（默认就是），也只有同样文本的SQL语句会集合在一起。实际中这会造成TKPROF很难对快速解析问题进行分析。但指定record选项可以使这个过程简单一些。这样的话，文件仅会包含生成的SQL语句。

```
tkprof <trace file> <output file> sys=no sort=prsela,exeela,fchela record=<sql file>
```

接着可以使用命令行工具如grep和wc来找出相似的SQL语句有多少条。例如，下面的命令返回的值是10 000：

```
grep "SELECT pad FROM t WHERE val =" <sql file> | wc -l
```

2. 使用TVD$XTAT

TVD$XTAT不需要指定特别的选项：

```
tvdxtat -i <trace file> -o <output file>
```

输出文件的分析从查看整体资源使用率配置文件开始。处理持续了14秒。这段时间中，43%的时间花在了等待客户端响应上，40%的时间用于CPU计算。这里的指标基本与上一部分介绍的一致。只有精度不同。在第一部分唯一附加的信息是准确给出了未说明用途的时间。

```
                               Total             Number of Duration per
Component                   Duration       %       Events        Event
--------------------------- -------- ------- --------- ------------
SQL*Net message from client    6.243  43.075    10,000        0.001
CPU                            5.862  40.444       n/a          n/a
unaccounted-for                2.364  16.309       n/a          n/a
SQL*Net message to client      0.024   0.168    10,000        0.000
latch: shared pool             0.000   0.002         5        0.000
log file sync                  0.000   0.002         1        0.000
--------------------------- -------- -------
Total                         14.494 100.000
```

仅观察汇总的非递归SQL语句，可以看到全部的处理操作只有单独一条SQL语句。这是TKPROF和TVD$XTAT之间明显的区别。实际上，TVD$XTAT识别类似的SQL语句，并合在一起记录到报告里。

```
Statement            Total            Number of Duration per
ID        Type    Duration       % Executions    Execution
--------- ------ -------- ------- ---------- ------------
#1        SELECT   12.130  83.689     10,000        0.001
#2        COMMIT    0.000   0.002          1        0.000
```

```
--------- ------ -------- -------
Total             12.130  83.691
```

根据没有递归语句的1号SQL语句执行统计信息，解析时间占用了处理时间的95%（5.705/6.009）。这清晰地表明数据库引擎除了解析没有做其他事情。TKPROF和TVD$XTAT的数据文件在执行统计信息上略微不同的是，TVD$XTAT在解析调用数旁显示未命中数（换句话说，硬解析）。

```
Call     Count Misses   CPU Elapsed PIO    LIO Consistent Current Rows
------- ------ ------ ----- ------- --- ------ ---------- ------- -----
Parse   10,000 10,000 5.548   5.705   0      0          0       0     0
Execute 10,000      0 0.176   0.156   0      0          0       0     0
Fetch   10,000      0 0.138   0.148   0 23,051     23,051       0 3,048
------- ------ ------ ----- ------- --- ------ ---------- ------- -----
Total   30,000 10,000 5.862   6.009   0 23,051     23,051       0 3,048
```

这些执行统计信息的问题是大约51%的响应时间不存在。不管怎样，你可以通过查看此处显示的SQL语句级别上的资源使用率配置文件，看到部分丢失的时间；特别是，等待客户端花费了6.243秒。

```
                              Total         Number of Duration per
Component                  Duration       %    Events        Event
-------------------------- -------- ------- --------- ------------
SQL*Net message from client   6.243  51.470    10,000        0.001
CPU                           5.862  48.327       n/a          n/a
SQL*Net message to client     0.024   0.201    10,000        0.000
latch: shared pool            0.000   0.003         5        0.000
-------------------------- -------- -------
Total                        12.130 100.000
```

3. 使用活动会话历史

活动会话历史基于采样。因此要执行明智的分析，需要大量的采样。考虑到测试案例1只执行了十几秒，使用活动会话历史并不能分析出准确的信息。本节的目的是向你展示查询的类型，你或许想要识别哪个SQL语句被解析以及它花费的时间。

在活动会话历史中，in_parse和in_hard_parse标志告诉你取样时会话是否在解析SQL语句。基于这些标志，可以针对某一会话写出类似下面的查询，来评估DB time和解析SQL语句花费的时间（请注意，这两个数字的单位都是秒）：

```
SQL> SELECT count(*) AS db_time,
  2         count(nullif(in_parse, 'N')) AS parse_time,
  3         count(nullif(in_hard_parse, 'N')) AS hard_parse_time
  4  FROM v$active_session_history
  5  WHERE session_id = 68
  6  AND session_serial# = 23;

DB_TIME PARSE_TIME HARD_PARSE_TIME
------- ---------- ---------------
      5          4               4
```

如果发现解析有问题（比如，根据上一个数据，80%的时间用于解析），你不仅需要知道具体解析的SQL语句，还要知道每条语句花费的时间。不幸的是，活动会话历史其中一个局限性就是并不能直接看到SQL文本。要获取语句文本，需要基于SQL ID来去查询另一个视图。例如，可以将v$active_

session_history联接到v$sqlarea。

不幸的是，游标保存在库缓存中的时间很短，尤其是因为快速解析而导致繁忙的数据库实例经历性能问题时。结果，你或许无法获取足够的信息。要想略微提高获取更多信息的可能性，可以使用v$sqlstats来代替v$sqlarea。实际上，前者有更大的保留期。例如，下面的查询能够取回与解析相关的四个采样中仅有的两条SQL语句（请记住，测试案例1执行了10 000条SQL语句）：

```
SQL> SELECT a.sql_id, s.sql_text, count(*) AS parse_time
  2  FROM v$active_session_history a, v$sqlstats s
  3  WHERE a.sql_id = s.sql_id(+)
  4  AND a.session_id = 68
  5  AND a.session_serial# = 23
  6  AND a.in_parse = 'Y'
  7  GROUP BY a.sql_id, s.sql_text
  8  ORDER BY count(*) DESC;

SQL_ID        SQL_TEXT                          PARSE_TIME
------------- --------------------------------- ----------
a6z6qamdcwqdv                                            1
2hcrrthw3w4y8                                            1
aydf9rbd6mz1m SELECT pad FROM t WHERE val = 9580         1
50m9q01tmghmw SELECT pad FROM t WHERE val = 7574         1
```

4. 总结问题

通过活动会话历史执行的分析对本例来说并不是特别有帮助。这是因为对单个会话在十几秒内进行采样只能得到几个样本。然而，TKPROF和TVD$XTAT执行的分析，清晰地展示了数据库引擎单独解析的处理信息。但是，在数据库端，解析只占了全部响应时间的39%（5.705/14.494）。

这代表排除它大于一半响应时间的可能。分析也显示了如下的10 000条SQL语句只解析并执行一次：

```
SELECT pad FROM t WHERE val = 0
```

由于使用的文字不断变化，在库缓存中的共享游标无法重用。换句话说，每个解析都是硬解析。图12-1图示了这个处理。

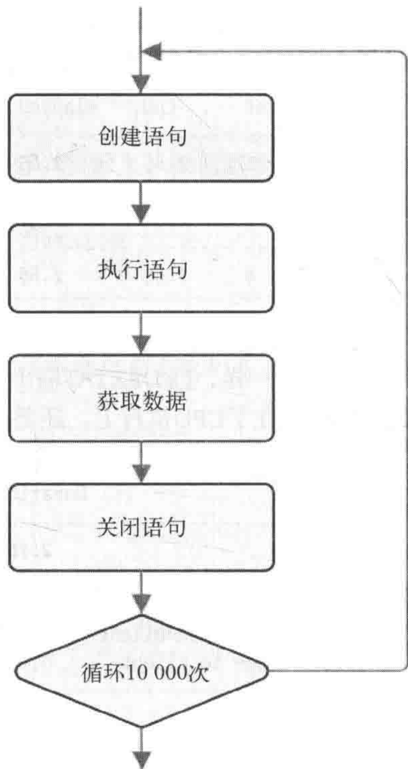


图12-1 测试案例1执行的处理

注意 图12-1显示的处理稍后会被当作测试案例1。

毫无疑问，这样的处理是低效的。请参考12.2节，找寻对应的解决方案。

12.1.2 长解析

接下来的部分将介绍如何识别长解析造成的性能问题。但是，不包含活动会话历史。原因很简单，解析调用很难看到超过几秒的。因此，大多数时候通过活动会话历史来分析这样的问题是不明智的。总之，如果遇到解析调用花费了几分钟的情况，那么使用活动会话历史进行分析的过程与12.1.1节介绍的类似。鉴于第3章介绍了两个探查器，TKPROF和TVD$XTAT，我会使用与它们的输出文件相同的例子。本节用到的示例跟踪文件是通过执行long_parse.sql脚本生成的。跟踪文件和输出文件可在long_parse.zip文件中找到。

1. 使用TKPROF

与快速解析一样，分析开始于TKPROF输出的尾部。在这个案例中，重点需要注意处理持续了大约2秒而应用仅执行了3条SQL语句。其他的SQL语句都是有数据库引擎递归调用的。

```
   1  session in tracefile.
   3  user SQL statements in trace file.
  13  internal SQL statements in trace file.
  16  SQL statements in trace file.
  16  unique SQL statements in trace file.
9644  lines in trace file.
   2  elapsed seconds in trace file.
```

通过查看输出文件中第一个SQL语句的执行统计信息，可以看出它不仅占用了全部的响应时间（超过2秒），而且所有的时间都用在单独一个解析上：

```
Call     count       cpu    elapsed       disk      query    current       rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      2.65       2.65          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        2      0.00       0.00         10         10          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      2.65       2.66         10         10          0           1
```

2. 使用TVD$XTAT

与快速解析一样，TVD$XTAT输出分析始于查看整体资源使用率配置文件。处理持续了2.8秒，其中98%的时间花在了CPU执行上。还要注意，在本例里，未说明时间非常短，因此完全可以忽略。

```
                              Total          Number of Duration per
Component                  Duration       %     Events        Event
-------------------------- -------- ------- --------- ------------
CPU                           2.769  98.383       n/a          n/a
db file sequential read       0.027   0.943       314        0.000
unaccounted-for               0.017   0.596       n/a          n/a
SQL*Net message from client   0.002   0.078         3        0.001
SQL*Net message to client     0.000   0.000         3        0.000
-------------------------- -------- -------
Total                         2.814 100.000
```

仅通过查看非递归SQL语句汇总，可以看到执行了3条SQL语句。其中一条SELECT语句占用了几乎全部的响应时间。

```
Statement            Total            Number of Duration per
ID        Type    Duration        % Executions    Execution
--------- ------ -------- ------- ---------- ------------
#1        SELECT    2.791  99.167          1        2.791
#9        PL/SQL    0.005   0.166          1        0.005
#12       PL/SQL    0.002   0.071          1        0.002
--------- ------ -------- -------
Total               2.797  99.404
```

根据导致该问题SQL语句的递归执行统计信息，单独一个解析操作占用了大约100%（2.654/2.661）的响应时间。这清晰地显示了数据库引擎除了解析没有做其他任何事。

```
Call    Count Misses   CPU Elapsed PIO LIO Consistent Current Rows
------- ----- ------ ----- ------- --- --- ---------- ------- ----
Parse       1      1 2.653   2.654   0   0          0       0    0
Execute     1      0 0.002   0.002   0   0          0       0    0
Fetch       2      0 0.004   0.006  10  10         10       0    1
------- ----- ------ ----- ------- --- --- ---------- ------- ----
Total       4      1 2.659   2.661  10  10         10       0    1
```

3. 总结问题

分析显示单个SQL语句占用了几乎全部的响应时间。此外，整个响应时间是用来解析这个SQL语句的。删除它也许会大大减少响应时间。

12.2 解决解析问题

解决解析问题很明显的方式是避免解析。但这并不总是可行。实际上，根据解析问题是与快速解析有关还是与长解析有关，你需要使用不同的技巧来解决问题。我会分别介绍这些技巧。12.1节中使用的例子将用作解释解决方案的基础。

注意 接下来的部分会通过性能测试的不同结果，来展示解析的影响。性能指标仅用来辅助对比不同的处理，使你更好地理解解析的影响。记住，每个系统和应用都有它们各自的特点。因此，根据环境不同，使用的技巧也会不同。

12.2.1 快速解析

本节介绍如何利用预处理语句来避免不必要的解析操作。鉴于执行细节跟开发环境有关，这里无法详细介绍。在本章稍后，特别是12.4节，我提供了关于PL/SQL、OCI、JDBC、ODP.NET和PHP的详细信息。

1. 使用预处理语句

当一个SQL语句使用不断变化的文字导致解析问题时，首先要做的是使用绑定变量替换文字。为此，你需要使用预处理语句（prepared statement）。使用预处理语句的目的是让所有相似的SQL语句（这

些语句之间只有绑定变量是否使用的文本差异）共享单个游标，以此来避免不必要的硬解析转换成软解析。图12-2图示了案例1提升测试性能的操作。

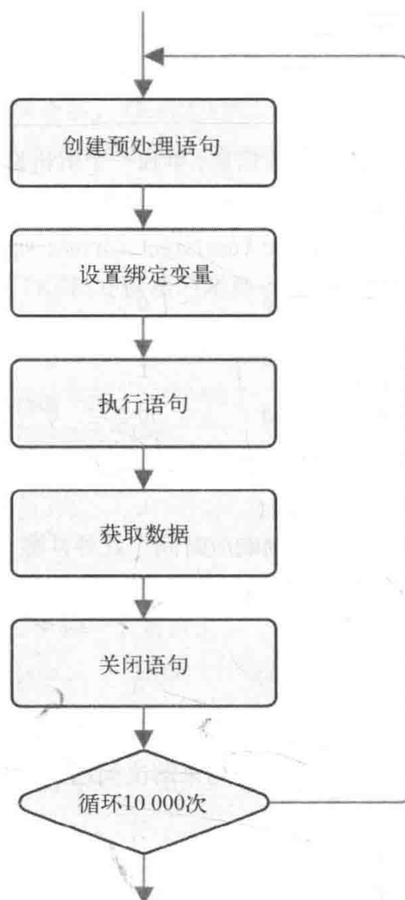


图12-2　测试案例2执行的操作

注意　图12-2显示的操作稍后将作为测试案例2。

正如图12-3所示，随着性能的提升，与测试案例1相比，测试案例2的响应时间下降了大约41%。这归功于预处理语句，因为新的代码只需要执行一次硬解析。因此，测试案例1里数据库引擎执行的大多数处理都可以避免。但是，请注意仍需执行10 000个软解析。

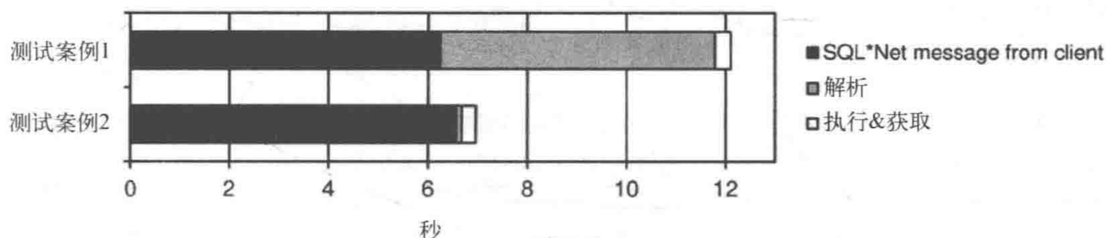


图12-3 对比测试案例1和测试案例2的数据库端资源使用率配置文件（占用少于1%响应时间的组件不会显示，因为它们不可见）

2. 重用预处理语句

上一节我建议使用预处理语句。重用它们可以更好地清除硬解析和软解析。由于测试案例2中解析的运行时间几乎可以忽略，你应该考虑一下原因。在给出答案之前，请查看重用单个预处理语句相关处理的性能指标。图12-4图示了测试案例2提升性能的操作。

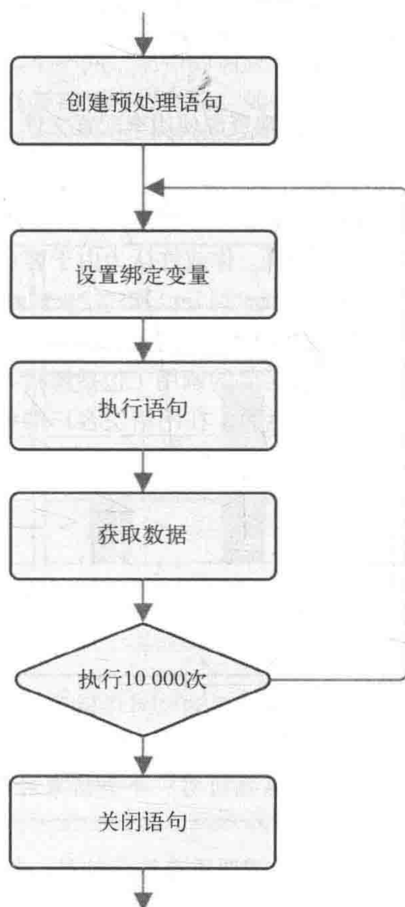


图12-4 测试案例3执行的操作

注意　图12-4显示的操作稍后将作为测试案例3。

正如图12-5所示，随着性能的提升，与测试案例1和测试案例2相比，测试案例3的响应时间分别下降了大约61%和33%。这归功于预处理语句，因为新的代码只需要执行一次硬解析。因此，测试案例1中数据库引擎执行的大多数处理都可以避免。但请注意仍需执行10 000个软解析。真正重要的区别并不在解析上的CPU time（在测试案例2中已经非常低了），而是对`SQL*Net message from client`等待的减少造成的。这代表你在网络或客户端节省了资源，也可能这两方面都节省了资源。

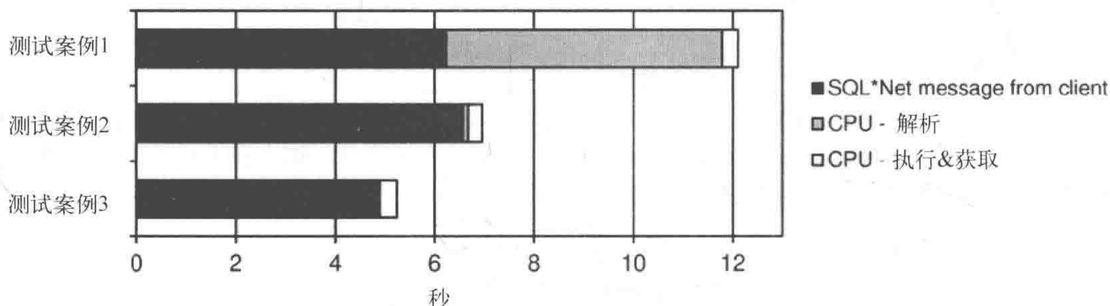


图12-5　对比三个测试案例的数据库端资源使用率配置文件（占用少于1%响应时间的组件不会显示，因为它们不可见）

在测试案例2中，数据库级别的软解析执行持续了大约十分之一秒。问题是，提升是从哪来的？肯定不是来自数据库级别上资源使用率的降低。你或许认为由于客户端与服务器端减少了通信而提高了性能。然而，通过查看`SQL*Net message from client`和`SQL*Net message to client`等待数，可以发现在这三个测试案例中并没有区别。在每个案例中，都是10 000次通信。显然这是因为完成了一万次执行，因此，这意味着在这个案例中，所有必要的调用（包括解析、执行以及提取的调用）都被客户端驱动程序打包到单条SQL*Net消息中了。然而，在网络层客户端和服务器端传输的消息大小不同。可以使用以下查询来获取关于它们的信息：

```
SELECT sn.name, ss.value
FROM v$statname sn, v$sesstat ss
WHERE sn.statistic# = ss.statistic#
AND sn.name LIKE 'bytes%client'
AND ss.sid = 42
```

注意　在SQL语句级别，无法检查客户端与服务器端之间传输的总数据量。因此，上一个查询取回的统计信息是会话级别上的。在这个案例中，这么做没有问题，因为我可以保证这个会话仅执行我的测试案例的SQL语句。此外，我通过另一个会话来对动态性能视图进行查询。

图12-6显示了三个测试案例的网络流量。需要重点注意的是，在测试案例2中，切换到预处理语句时，由数据库引擎接收到的信息大小略有增加。将测试案例3与其他两个测试案例相比，最重要的区别是大量减少数据库引擎接收和发送信息的大小。这是因经过网络发送的数据为了打开和关闭新游标

（在测试案例3中，SQL语句的文本通过网络发送，连同打开游标只执行一次，而游标也在最后仅关闭一次）而引起的。

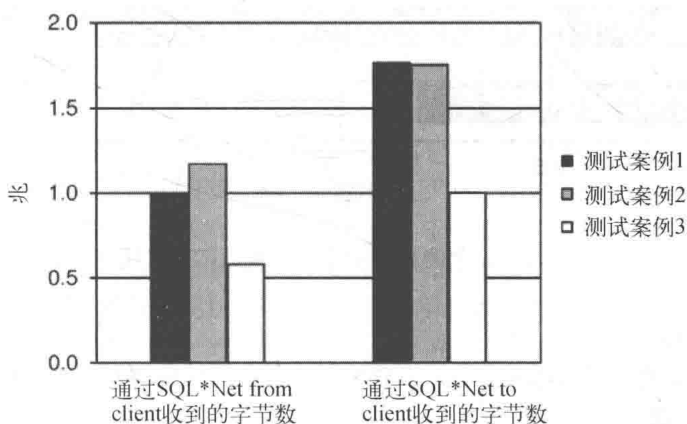


图12-6 三个测试案例中单个执行的网络流量

由于通过网络发送的信息大小不同，期望的响应时间会基于网络速度。如果网络快，客户端与服务器端之间的通信影响就小，或者甚至可以忽略。如果网络慢，影响会很大。图12-7显示了两种网络速度的响应时间。显然数据库引擎处理测试案例的调用时间并不依赖网络速度。

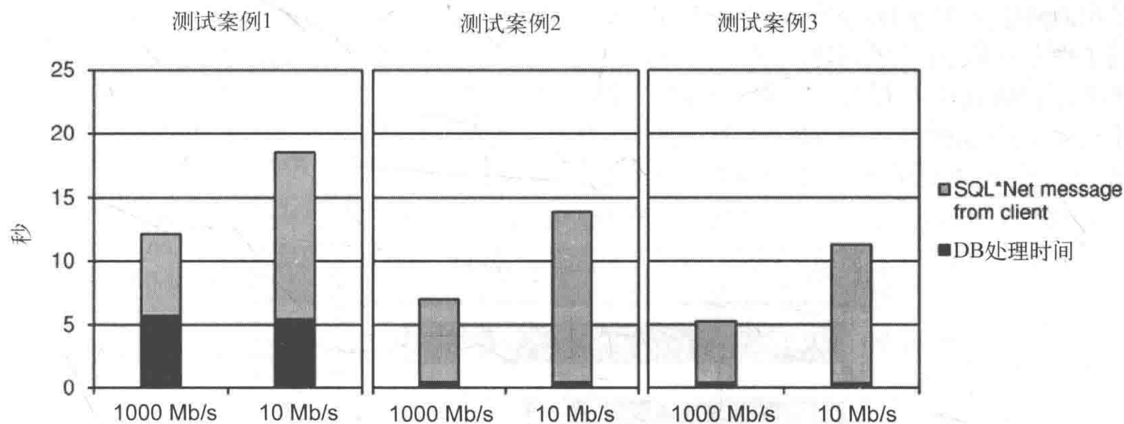


图12-7 三个测试案例在两种网速下的响应时间

即使网络速度对整体的响应时间影响非常大，需要重点注意的还是三个测试案例对客户端资源的影响，特别是使用不同CPU的影响。图12-8显示了三个测试案例中客户端的CPU使用率。测试案例1和测试案例2的指标对比显示出，使用绑定变量会对客户端造成开销。测试案例2和测试案例3的指标对比显示出，创建和关闭SQL语句也会对客户端造成开销。

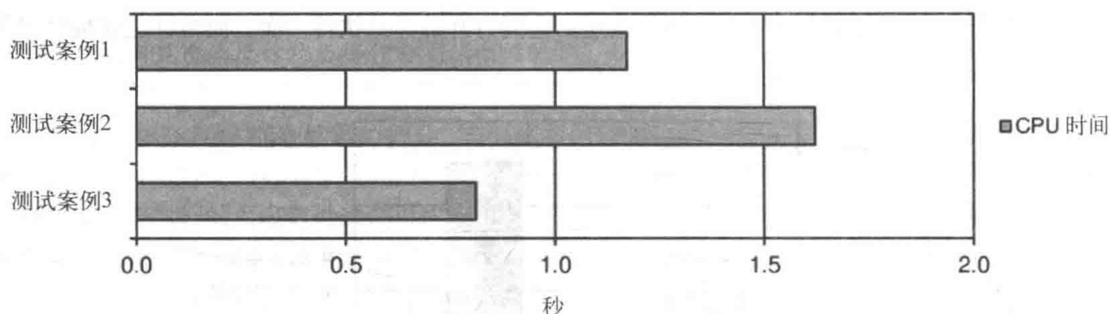


图12-8　三个测试案例中客户端的CPU使用率

3. 客户端语句缓存

如果应用打开和关闭太多游标而导致太多软解析，引起了性能问题，就可以使用该特性来解决。测试案例2就是这个问题。

客户端语句缓存的概念非常简单。每当应用关闭一个游标，代替真实的关闭，客户端数据库层（负责与数据库引擎之间的通信）会保持它打开，并将其添加到缓存中。接着，稍后如果再次打开和解析基于同样SQL语句的游标，代替真正的打开和解析，会重用客户端缓存的游标。因而不会发生软解析。基本上，目标是使应用的行为像测试案例3一样，即使它的写法很像测试案例2。

要使用这个特性，通常仅需要应用它并定义可缓存的最大游标数。请注意当缓存满了时，最近最少使用的游标会被关闭并替换成新游标。应用新增初始化代码或在环境中设置变量时会触发激活。它如何工作完全取决于开发环境。本章稍后，特别是在12.4节，会提供关于PL/SQL、OCI、JDBC、ODP.NET和PHP的详细信息。要设置最大缓存游标数，需要了解使用的应用。如果不知道，应该分析它并找出属于高软解析数的SQL语句数量。但这仅是首次估算。之后，仍需要执行一些测试来确保设置了正确的值。总之，它不应该超过初始化参数`open_cursors`的值。

正如图12-9所示，使用客户端语句缓存的测试案例2几乎与测试案例3一样。准确地说，它们都执行了一个硬解析和一个软解析。因此，多亏了语句缓存，客户端处理大幅度下降。

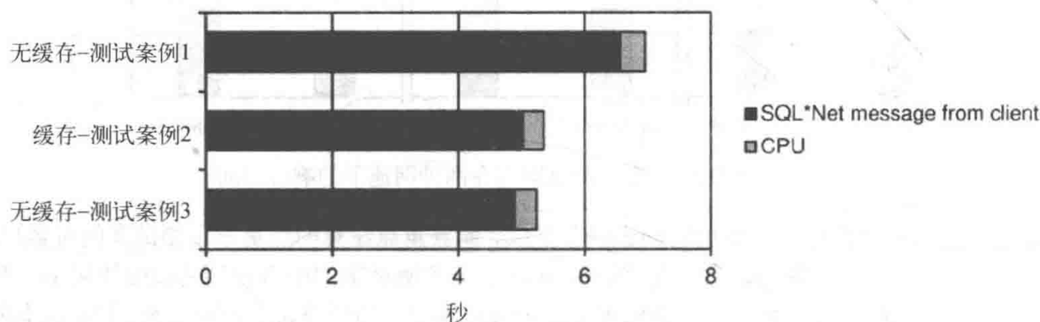


图12-9　对比使用和不使用客户端语句缓存的数据库端资源使用率配置文件（占用少于1%响应时间的组件不会显示，因为它们不可见）

4. 总结

通过利用带绑定变量的预处理语句来避免不必要的硬解析有时很重要。然而，当使用它们时，你应该能预料到会在客户端的CPU使用率和网络流量上有小的开销。你可以证明这个开销会造成性能问题，因此预处理语句和绑定变量应该仅在必要时使用。既然开销几乎可以忽略不计，那么最佳实践就是，只要它们没有导致低效的执行计划（更多详细信息，请参考第2章），就应该尽可能使用预处理语句和绑定变量。每当预处理语句频繁使用时，就应该重用它。这么做不仅可以避免软解析，同时也可以降低客户端CPU使用率和减少网络流量。唯一的问题是，一个预处理语句要保持打开状态就会使用客户端和服务器端更多的内存。这代表当每个会话保持上千个游标打开时需要谨慎处理，并且只有在内存足够时才使用。同样需要注意，初始化参数open_cursors限制了单个会话同时打开游标的数量。万一缓存了许多预处理语句，最好是使用客户端语句缓存并仔细设置缓存大小，而不是手动保持它们打开。这样的话，通过允许有限的预处理语句缓存，内存压力或许会减轻。

12.2.2 长解析

如果长解析只执行了几次（或者如上一个例子，只有一次），通常无法避免长解析。实际上，SQL语句必须至少解析一次。此外，如果SQL语句很少执行，那么通常必然会进行硬解析，因为在各次执行之间游标会因过期而从库缓存中交换出来，尤其是在没有使用绑定变量的时候。因此，唯一可能的解决方案就是减少它自身的解析时间。

是什么导致长解析时间？通常，是由于查询优化器评估了太多不同的执行计划。此外，也可能是由于执行递归调用时正在进行动态采样。解决后者的方法很明显：要么降低动态采样级别，要么彻底禁用它。然而，解决前者会有些麻烦。实际上，要缩短解析时间，必须减小评估执行计划的数量。这通常可以通过hint或存储概要来强制使用某个执行计划。例如，在12.1节的例子中，SQL语句创建存储概要后，解析时间缩短了6倍（请查看图12-10）。通过直接指定hint在SQL语句中也可以达到同样的效果，当然前提是你可以修改代码。

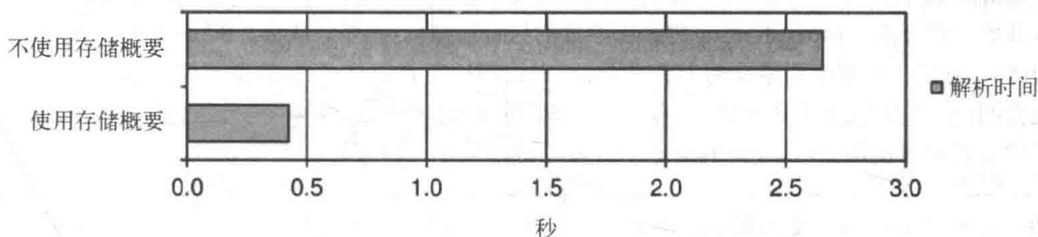


图12-10 使用和不使用存储概要的解析时间对比

12.3 避开解析问题

之前的章节描述了三个与快速解析有关的测试案例。第一个是一个糟糕代码书写的例子。第二个比第一个要好得多。第三个在大多数情况下是最好的一个。目前的窘境是类似于测试案例1这样的代码必须修改，才可以改善性能，但遗憾的是，这并非总是可行的。这是因为要么是代码不可用，技术性壁垒阻止增强代码（例如，预编译语句在编程环境中不可用），要么是实现所有必要的修改太过“昂贵”。

接下来的部分会介绍如何处理这样的问题，来使性能接近不存在解析问题的应用的性能。这样的方案即使性能不能与正确的实现一样好，但在某些情况下也比什么都不做要好。

注意　下面的小节通过展示不同的性能测试结果来描述解析的影响。性能指标只用来辅助对比不同的处理，使你更好地理解解析的影响。记住，每个系统和应用都有它们各自的特点。因此，根据环境不同，使用的技巧也会不同。

12.3.1　游标共享

这项功能是用来解决由应用程序未使用绑定变量引起的性能问题，因为这样做会导致过多的硬解析。在本章前面的部分，我在测试案例1中指出过这个问题。

游标共享（cursor sharing）的概念很简单。如果一个应用程序执行包含文字的SQL语句，并且游标共享处于启用状态，那么数据库引擎会自动使用绑定变量替换文字。这样，对于只有文字不同的SQL语句来说，硬解析可能会转为软解析。基本上，目标是让一个应用程序表现得与测试案例2类似，即使它的写法与测试案例1类似。

注意　游标共享不会替换通过PL/SQL执行的静态SQL语句中的文字。对于动态SQL语句来说，只有当字面值不会与绑定变量混淆的时候才会发生替换。这不是一个bug，而是一项设计决策。可以使用`cursor_sharing_mix.sql`脚本来重现这种行为。

游标共享是通过`cursor_sharing`初始化参数控制的。如果设置为`exact`，该特性会被禁用。换句话说，只有当SQL语句的文本完全相同时，它们才会共享父游标。如果将`cursor_sharing`设置为`force`或`similar`，则会启用该特性。默认值是`exact`。可以在系统和会话级别上修改它。也可以在SQL语句级别上通过指定`cursor_sharing_exact`提示来显式禁用游标共享。

Oracle Support文档1169017.1（*Deprecating the cursor_sharing =* ‘*SIMILAR*’ *setting*）显示，从11.1版本开始，将废弃`cursor_sharing`初始化参数的`similar`值。此外，从11.2.0.3版本开始，将这个参数设置为`similar`时，数据库引擎会将其作为`force`来处理！废弃值`similar`的主要原因有两个。第一，你很快就会明白，它的实现存在问题。第二，自适应游标共享（该特性的相关信息请参考第2章）的引入使得没有必要再使用`similar`。事实上，自适应游标共享可以在游标共享设置为`force`时起作用。

警告　游标共享以不稳定而闻名。这是因为，经过这些年，找到并修复了与之相关的大量bug。因此，如果你正在考虑使用它，我的建议是仔细查阅Oracle Support文档94036.1（*Init.ora Parameter* “*CURSOR_SHARING*” *Reference Note*），尤其是已知bug列表。

鉴于游标共享可以通过两个值来启用，`force`和`similar`，我们讨论一下其中的区别。出于这个目的，会在10.2.0.5版本的数据库上分别使用不同的`cursor_sharing`值执行测试案例1。我们来看一下使用值`force`时的结果。如图12-11所示，测试案例1（使用值`force`）中的数据库端资源使用率配置文件与测试案例2中的类似。实际上，它们两个都执行了一次单独的硬解析和9999次软解析。结果，多亏有了游标共享，解析时间大幅降低了。使用值`force`，只是在CPU使用率上有轻微的改善。因为数据库

引擎为了使用绑定变量替换字面值而必须执行更多的工作，但这是在预料之中的。

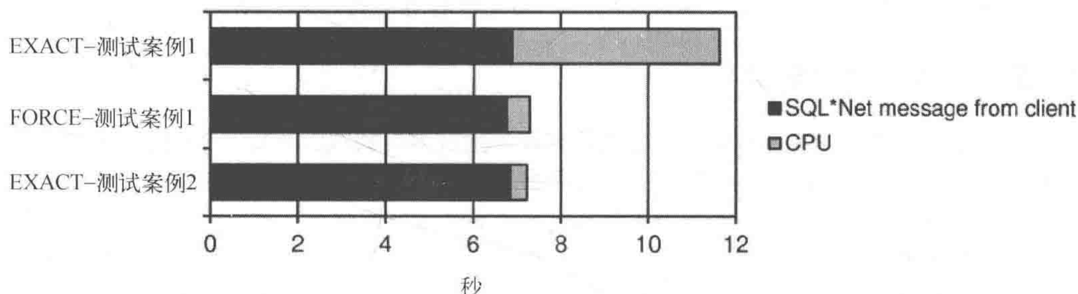


图12-11　将游标共享设置为force时对比数据库端资源使用率配置文件（这里不会显示响应时间不足1%的那些组件）

如果不考虑自适应游标共享，值force相关的问题就变成，替换文字的所SQL语句会共享相同的文本来使用单个子游标。因此，文字（它对于直方图来说很关键）只有在与第一条提交的SQL语句关联的执行计划生成时会被扫视到。当后续的执行计划中使用的文字可能需要不同的执行计划时，这会导致性能问题。为了避免这种情况的发生，可以使用值similar。事实上，使用similar，在重用一个已经可用的的游标之前，SQL引擎会检查其中一个被替换的文字是否有对应的直方图存在。如果不存在，则可以使用任何有兼容执行环境的子游标。如果确实存在，则只有使用一样文字创建的子游标才可以被使用。结果，使用similar会针对每一个文字值使用单独的游标（使用更少的内存），这会代替针对每个文字值使用单独的父游标。

如图12-12所示，在测试案例1中，使用值similar的数据库端资源使用率配置文件要表现得比使用值exact还要糟。问题不仅是执行了10 000次硬解析，由于游标共享，这样的解析操作的CPU使用率也会更高。事实上，解析时间会随着每个父游标的子游标数量直线上升。解析时间直线上升，是因为在解析期间，SQL引擎必须检查是否有可用的子游标可以重用。因此，必须扫描子游标的列表并探测每个子游标的兼容性。简单来说，过多的子游标抑制了良好的性能。注意在替换完文字值后，所有SQL语句拥有相同的文本。因此，库缓存包含着单独一个父游标，该父游标拥有成千的子游标。

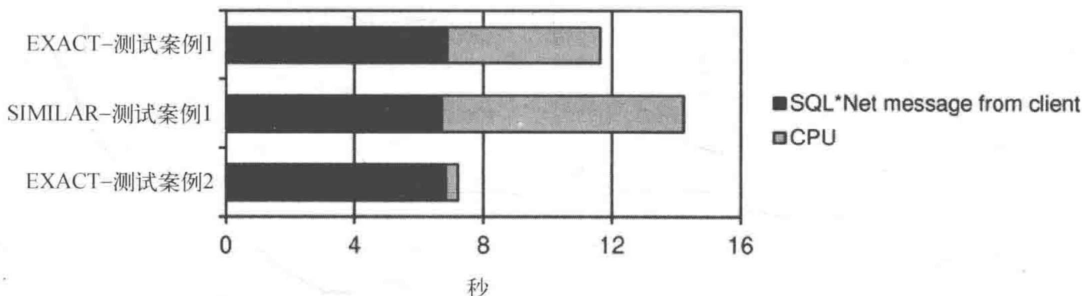


图12-12　将游标共享设置为similar时比较数据库端资源使用率配置文件（这里不会显示响应时间不足1%的那些组件）

总之，如果一个应用程序使用文字值并且将游标共享设置为similar，其性能取决于是否存在相关

直方图。如果它们存在，similar的表现就与exact类似。如果它们不存在，similar表现就与force类似。这意味着如果面临解析问题，通常使用similar是没有意义的。

12.3.2 服务器端语句缓存

这个功能与客户端语句缓存类似，它是用来在发生过多的软解析时减少负载的。从概念上来看，这两种类型的语句缓存是类似的，除了一个在服务器端实现，另一个在客户端实现。从性能的角度来看，差别还是很大的。事实上，服务器端实现远不及客户端实现强大。这是因为服务器端实现仅在服务器端减少软解析的负载，而多数情况下，客户端软解析的负载远比服务器端要大。实现服务器端语句缓存的唯一优势，是可以在数据库引擎中缓存部署的PL/SQL或Java代码执行的SQL语句。

如果一个应用程序执行大量的软解析，在库缓存的闩和互斥上过高的压力也会导致在数据库引擎上出现显著的争用。下面的数据库端资源使用率配置文件展示了这样情况。当数据库引擎为相同的SQL语句每秒处理超过30 000次解析时，启动了测试案例2。尽管这肯定不是一个正常的负载，但它有助于证实服务器端游标缓存的影响。

```
                               Total          Number of Duration per
Component                   Duration       %     Events        Event
--------------------------- -------- ------- ---------- ------------
SQL*Net message from client    4.166  54.569     10,000        0.000
library cache: mutex X         2.622  34.339        158        0.017
CPU                            0.557   7.294        n/a          n/a
latch free                     0.265   3.473          1        0.265
SQL*Net message to client      0.014   0.177     10,000        0.000
cursor: pin S                  0.011   0.148          1        0.011
--------------------------- -------- -------
Total                          7.635 100.000
```

一旦服务器端软解析的负载影响性能而又无法修改应用程序时，服务器端语句缓存可能会有帮助。这个例子中，在启用并施加相同的负载后，产生的资源使用率配置文件如下所示。注意关联到库缓存的闩锁和互斥上的大部分等待已消失。

```
                               Total         Number of Duration per
Component                   Duration       %    Events        Event
--------------------------- -------- ------- --------- ------------
SQL*Net message from client    4.646  85.959     10000        0.000
CPU                            0.420   7.769       n/a          n/a
cursor: pin S                  0.328   6.070         2        0.164
SQL*Net message to client      0.011   0.202     10000        0.000
--------------------------- -------- -------
Total                          5.405 100.000
```

服务器端语句缓存是通过session_cached_cursors初始化参数配置的。它的值指定每个会话能够缓存的最大游标数量。所以如果将它设置为0，就会禁用该特性。反之如果设置的值大于0，则会启用该特性。在10.2版本中，默认值是20；从11.1版本开始，默认值是50。在系统级别上，只有重启实例才能改变它。在会话级别上，可动态修改该参数。与客户端语句缓存一样，要决定缓存游标的最大数量，需要了解正在使用的应用程序，或分析以找出有多少SQL语句产生了大量软解析。然后，根据这个初步的估算，有必要进行一些测试来验证这个值是否合适。在这些测试期间，要验证缓存是否有效，可以通过验证对响应时间的影响，也可以通过从下面的查询中查看统计结果来验证。注意，在系统级别上也有相同的统计

信息可用。无论如何，应该关注单独一个存在负债问题的会话，以便找出可用的线索。

```
SQL> SELECT sn.name, ss.value
  2  FROM v$statname sn, v$sesstat ss
  3  WHERE sn.statistic# = ss.statistic#
  4  AND sn.name IN ('session cursor cache hits',
  5                  'session cursor cache count',
  6                  'parse count (total)')
  7  AND ss.sid = 42;

NAME                                VALUE
-------------------------- ----------
session cursor cache hits            9997
session cursor cache count              9
parse count (total)                 10008
```

第一，将缓存游标的数量（session cursor cache count）与session_cached_cursors初始化参数的值进行对比。如果前者小于后者，则意味着增加该初始化参数的值应该对缓存的游标数量没有影响。否则，如果两个值相等，增加该初始化参数的值可能有助于缓存更多的游标。无论如何，超过open_cursors初始化参数的值都是没意义的。例如，根据上面的统计信息，现在缓存中有九个游标。因为测试期间session_cached_cursors初始化参数被设置为50，增加它的值就没有用处。

第二，使用这些附加的指标，可以检查相对于解析调用的总数（parse count (total)），有多少解析调用是通过服务器端语句缓存优化的（session cursor cache hits）。如果两个值接近，可能并不值得花时间增加缓存的大小。在上面这些统计信息中，因为缓存避免了超过99%（9997/10 008）的解析，所以增加它很可能没什么意义。

小心Bug!

由parse count (total)和session cursor cache hits统计信息提供的值经常会引起几个bug。这些bug中你最可能碰到的是以下这些。

- 从11.1.0.6版本开始，session cursor cache hits统计信息会为利用PL/SQL客户端语句缓存的游标增加数值。因此，session cursor cache hits统计信息可能会比parse count (total)统计信息高出许多。默认情况下会将客户端语句缓存在PL/SQL程序中。因此，使用PL/SQL 时，session cursor cache hits统计信息就变得没用了。
- 从11.2.0.1版本开始，会话级别上的session cursor cache hits统计信息存储在一个占用16位的无符号整数中。因此，命中超过65 535次的会话会溢出，并且值会从0重新开始。而且，即使这个统计信息在系统级别没有这样的限制，在会话级别的溢出仍然会引起系统级别的统计信息减少65 535。结果，session cursor cache hits统计信息在系统和会话级别上几乎没有用处。
- 在11.2.0.3版本中，parse count (total)统计信息并没有为使用服务器端语句缓存的游标增加数据。结果，session cursor cache hits统计信息要比parse count (total)统计信息的值高得多。由于默认会使用服务器端语句缓存，实际上parse count (total)统计信息在11.2.0.3版本中没有实际用处。该bug在11.2.0.4版本中已修复。

综上所述，当依赖并解释session cursor cache hits统计信息时，要十分小心。回顾已知bug来确保没有适用于你的情况，特别是要记住这里列出的三个bug。

在之前的统计信息中还有一件重点需要注意的事情，缓存中“只有”9997次命中。既然测试案例2执行了相同的SQL语句10 000次，为什么不是9999？答案是一个游标只有在它已经被执行多次的情况下才会被放入游标缓存中。这么做的原因是防止缓存那些只执行一次的游标。获得9999这个数，只能是在第一次解析调用之前就已经有一个可共享游标存在于库缓存中。

总之，服务器端语句缓存是一个重要的功能。事实上，如果正确设置大小，它可能会节省一些服务器端的负载。然而，不能因为这个功能，应用就找借口不再管理游标。这是因为正如你上面看到的，当缓存在服务器端执行而不是客户端执行时，解析的负载会更高。

12.4　使用应用编程接口

本节的目标是描述与为不同的应用编程接口解析相关的功能。在之前章节的描述中，为了避免不必要的硬解析和软解析，应该有三个关键的功能可以使用：绑定变量、重用语句以及客户端语句缓存。表12-1总结了在不同的应用编程接口中这些特性的可用情况。接下来的小节会为PL/SQL、OCI、JDBC、ODP.NET以及PHP提供一些详细的信息。

表12-1　由不同的应用编程接口提供的功能概览

应用编程接口	绑定变量	语句重用	客户端语句缓存
Java 数据库连接 (JDBC)			
java.sql.Statement			
java.sql.PreparedStatement	√	√	√
Oracle Call Interface (OCI)	√	√	√
Oracle C++ Call Interface (OCCI)	√	√	√
Oracle Data Provider for .NET (ODP.NET)	√		√
Oracle Objects for OLE (OO4O)	√		
Oracle Provider for OLE DB	√		√
PHP (PECL OCI8 扩展)	√	√	√
PL/SQL			
静态 SQL	√		√
本地动态SQL(EXECUTE IMMEDIATE)	√		√
本地动态SQL(OPEN/FETCH/CLOSE)	√		
使用dbms_sql包的动态SQL	√	√	
预编译程序	√	√	√
SQLJ	√		√

12.4.1　PL/SQL

PL/SQL提供了不同的方法来执行SQL语句。主要的两个类别是静态SQL和动态SQL。动态SQL能够进一步分成三个子类别：EXECUTE IMMEDIATE、OPEN/FETCH/CLOSE以及dbms_sql包。唯一与解析相关的功能是它们都可以使用绑定变量。而实际上，只有重用语句和缓存客户端语句的部分可以使用。它们并不是对所有SQL语句类别都有效。接下来的小节将会描述这四种类别的每个细节。

注意 鉴于PL/SQL是在数据库引擎中运行的，讨论客户端语句缓存好像有点奇怪。其实，从SQL引擎的视角来看，PL/SQL引擎就是一个客户端。在这个客户端中，客户端语句缓存的概念将在这里实现。

在本节例子中提供的PL/SQL代码块来自ParsingTest1.sql、ParsingTest2.sql以及ParsingTest3.sql脚本，分别实现测试案例1、2和3。

1. 静态SQL

静态SQL被集成到PL/SQL语言中。就像它的名称一样它是静态的，因此，在PL/SQL编译期间SQL语句必须是完全已知的。出于这个原因，如果一条SQL语句引用了PL/SQL变量，则不可避免地要使用绑定变量。例如，不可能使用静态SQL写出一段代码来重现测试案例1。

编写静态SQL有两种方式。第一种是基于隐式游标，但它没有控制游标生命周期的能力。下面的PL/SQL代码块实现了测试案例2：

```
DECLARE
  l_pad VARCHAR2(4000);
BEGIN
  FOR i IN 1..10000
  LOOP
    SELECT pad INTO l_pad
    FROM t
    WHERE val = i;
  END LOOP;
END;
```

第二种方式是基于显式游标。在这种情况下，可以对游标进行某些控制。不管怎样，打开/解析/执行阶段被合并成一个单独的操作（OPEN）。这意味着仅可以控制提取和关闭阶段。下面的PL/SQL代码块实现了测试案例2：

```
DECLARE
  CURSOR c (p_val NUMBER) IS SELECT pad FROM t WHERE val = p_val;
  l_pad VARCHAR2(4000);
BEGIN
  FOR i IN 1..10000
  LOOP
    OPEN c(i);
    FETCH c INTO l_pad;
    CLOSE c;
  END LOOP;
END;
```

尽管这两种方式都防止了不良代码（测试案例1），但它们也不允许写出特别高效的代码（测试案例3）。这是因为没有完全控制游标。但从性能的角度看，这两种方法是类似的。

为了解决这个问题，可以使用客户端语句缓存。缓存游标的最大数量由session_cached_cursors初始化参数决定。在10.2版本中，默认的缓存游标数量是20，而从11.1版本开始是50。这个初始化参数，并不与客户端语句缓存直接相关，而是“错误”地配置了它！事实上，这与用于控制服务器端语句缓存的初始化参数是同一个。

2. 本地动态SQL：EXECUTE IMMEDIATE

从游标管理的角度来看，基于EXECUTE IMMEDIATE的本地动态SQL与使用隐式游标的静态SQL类似。换句话说，它不能控制游标的生命周期。下面的PL/SQL代码块实现了测试案例2：

```
DECLARE
  l_pad VARCHAR2(4000);
BEGIN
  FOR i IN 1..10000
  LOOP
    EXECUTE IMMEDIATE 'SELECT pad FROM t WHERE val = :1' INTO l_pad USING i;
  END LOOP;
END;
```

没有了对游标的控制，不可能写出实现测试案例3的代码。出于这个原因，可以像静态SQL那样使用客户端游标缓存。

3. 本地动态SQL：OPEN/FETCH/CLOSE

从游标管理的角度来看，基于OPEN/FETCH/CLOSE的本地动态SQL与使用隐式游标的静态SQL类似。换句话说，它仅能控制提取（FETCH）阶段。下面的PL/SQL代码块实现了测试案例2：

```
DECLARE
  TYPE t_cursor IS REF CURSOR;
  l_cursor t_cursor;
  l_pad VARCHAR2(4000);
BEGIN
  FOR i IN 1..10000
  LOOP
    OPEN l_cursor FOR 'SELECT pad FROM t WHERE val = :1' USING i;
    FETCH l_cursor INTO l_pad;
    CLOSE l_cursor;
  END LOOP;
END;
```

没有对游标的完全控制，不可能写出实现测试案例3的代码。此外，使用基于OPEN/FETCH/CLOSE的动态SQL，数据库引擎无法利用客户端语句缓存。这意味着要解决这种代码引起的解析问题的唯一途径，是使用EXECUTE IMMEDIATE或dbms_sql包对语句进行改写。作为一种变通方案，还可以考虑服务器端语句缓存。

4. 本地动态SQL：dbms_sql包

dbms_sql包提供对游标的生命周期的完全控制。在下面的PL/SQL代码块中（测试案例2），请注意显式编码的每一步：

```
DECLARE
  l_cursor INTEGER;
  l_pad VARCHAR2(4000);
  l_retval INTEGER;
BEGIN
  FOR i IN 1..10000
  LOOP
    l_cursor := dbms_sql.open_cursor;
    dbms_sql.parse(l_cursor, 'SELECT pad FROM t WHERE val = :1', 1);
    dbms_sql.define_column(l_cursor, 1, l_pad, 10);
```

```
    dbms_sql.bind_variable(l_cursor, ':1', i);
    l_retval := dbms_sql.execute(l_cursor);
    IF dbms_sql.fetch_rows(l_cursor) > 0
    THEN
      NULL;
    END IF;
    dbms_sql.close_cursor(l_cursor);
  END LOOP;
END;
```

因为可以完全控制游标，实现测试案例3就没有问题了。下面的PL/SQL代码块展示了这样的例子。注意，为了避免不必要的软解析，准备游标（open_cursor、parse、define_column）和关闭游标（close_cursor）的过程被放置到循环外。

```
DECLARE
  l_cursor INTEGER;
  l_pad VARCHAR2(4000);
  l_retval INTEGER;
BEGIN
  l_cursor := dbms_sql.open_cursor;
  dbms_sql.parse(l_cursor, 'SELECT pad FROM t WHERE val = :1', 1);
  dbms_sql.define_column(l_cursor, 1, l_pad, 10);
  FOR i IN 1..10000
  LOOP
    dbms_sql.bind_variable(l_cursor, ':1', i);
    l_retval := dbms_sql.execute(l_cursor);
    IF dbms_sql.fetch_rows(l_cursor) > 0
    THEN
      NULL;
    END IF;
  END LOOP;
  dbms_sql.close_cursor(l_cursor);
END;
```

使用dbms_sql包时，数据库引擎无法利用客户端语句缓存。所以，为了优化一个有太多软解析的应用程序（如测试案例2），必须修改它以重用游标（如测试案例3）。作为一个权宜方案，可以考虑服务器端语句缓存。

12.4.2 OCI

OCI是一种低级别的应用编程接口。因此，它可以提供对游标生命周期的完全控制。例如，在下面的代码段中，实现了测试案例2，请注意显式编码的步骤：

```
for (i=1 ; i<=10000 ; i++)
{
  OCIStmtPrepare2(svc, (OCIStmt **)&stm, err, sql, strlen(sql), NULL, 0, OCI_NTV_SYNTAX,
                  OCI_DEFAULT);
  OCIDefineByPos(stm, &def, err, 1, val, sizeof(val), SQLT_STR, 0, 0, 0, OCI_DEFAULT);
  OCIBindByPos(stm, &bnd, err, 1, &i, sizeof(i), SQLT_INT, 0, 0, 0, 0, 0, OCI_DEFAULT);
  OCIStmtExecute(svc, stm, err, 0, 0, 0, 0, OCI_DEFAULT);
  if (r = OCIStmtFetch2(stm, err, 1, OCI_FETCH_NEXT, 0, OCI_DEFAULT) == OCI_SUCCESS)
  {
    // 对数据进行某些处理
```

```
  }
  OCIStmtRelease(stm, err, NULL, 0, OCI_DEFAULT);
}
```

既然可以对游标进行完全控制，也就可以实现测试案例3。下面的代码片段就是一个例子。注意，为了避免不必要的软解析，准备游标（OCIStmtPrepare2 和 OCIDefineByPos）和关闭游标（OCIStmtRelease）的函数被放置到循环外。

```
OCIStmtPrepare2(svc, (OCIStmt **)&stm, err, sql, strlen(sql), NULL, 0, OCI_NTV_SYNTAX,
                OCI_DEFAULT);
OCIDefineByPos(stm, &def, err, 1, val, sizeof(val), SQLT_STR, 0, 0, 0, OCI_DEFAULT);
for (i=1 ; i<=10000 ; i++)
{
  OCIBindByPos(stm, &bnd, err, 1, &i, sizeof(i), SQLT_INT, 0, 0, 0, 0, 0, OCI_DEFAULT);
  OCIStmtExecute(svc, stm, err, 0, 0, 0, 0, OCI_DEFAULT);
  if (r = OCIStmtFetch2(stm, err, 1, OCI_FETCH_NEXT, 0, OCI_DEFAULT) == OCI_SUCCESS)
  {
    // 对数据进行某些处理
  }
}
OCIStmtRelease(stm, err, NULL, 0, OCI_DEFAULT);
```

OCI不仅启用对游标的完全控制，而且还支持客户端语句缓存。要使用它，仅需启用语句缓存并使用OCIStmtPrepare2和OCIStmtRelease函数（如上面的例子那样）。调用OCIStmtRelease函数时，会将游标添加到缓存中。然后，通过OCIStmtPrepare2函数创建新的游标时，就会访问缓存以查找是否有一条拥有相同文本的SQL语句在其中。

启用语句缓存的方法有多种。基本上，只需要在会话打开或从一个池中恢复时指定它就可以。例如，如果通过OCILogon2函数打开一个没有保存到池中的会话，有必要指定OCI_LOGON2_STMTCACHE这个值来启用这种模式。

```
OCILogon2(env, err, &svc, username, strlen(username), password, strlen(password),
          dbname, strlen(dbname), OCI_LOGON2_STMTCACHE)
```

默认情况下，缓存的大小是20。下面的代码片段展示如何通过设置服务上下文上的OCI_ATTR_STMTCACHESIZE属性，将缓存的大小更改为50。注意，将这个属性设置为0会禁用语句缓存。

```
ub4 size = 50;
OCIAttrSet(svc, OCI_HTYPE_SVCCTX, &size, 0, OCI_ATTR_STMTCACHESIZE, err);
```

本节中提供的C代码的例子分别摘录自实现了测试案例1、2和3的ParsingTest1.c、ParsingTest2.c以及 ParsingTest3.c的文件。

12.4.3 JDBC

java.sql.Statement是由JDBC提供的执行SQL语句的基础类。如表12-1所示，使用它时出现解析问题并非不可能。事实上，它不支持绑定变量、游标的重用以及客户端语句缓存。基本上，使用它仅可能实现测试案例1。下面的代码片段进行了示范：

```
sql = "SELECT pad FROM t WHERE val = ";
for (int i=0 ; i<10000; i++)
{
```

```
  statement = connection.createStatement();
  resultset = statement.executeQuery(sql + Integer.toString(i));
  if (resultset.next())
  {
    pad = resultset.getString("pad");
  }
  resultset.close();
  statement.close();
}
```

为了避免由上面的代码片段执行所产生的硬解析，必须使用java.sql.PreparedStatement类（或者它的一个子类），它是java.sql.Statement的子类。下面的代码片段展示了使用它实现测试案例2。注意，用于查找的值是通过绑定变量定义的（在Java中使用一个问号定义并称作占位符），而不是循环传递给sql变量（如上面例子那样）。

```
sql = "SELECT pad FROM t WHERE val = ?";
for (int i=0 ; i<10000; i++)
{
  statement = connection.prepareStatement(sql);
  statement.setInt(1, i);
  resultset = statement.executeQuery();
  if (resultset.next())
  {
    pad = resultset.getString("pad");
  }
  resultset.close();
  statement.close();
}
```

接下来的改进还要避免软解析，换言之，实现测试案例3。如下面的代码片段所示，可以通过将创建和关闭预编译语句的代码移动到循环外面来实现这个目标：

```
sql = "SELECT pad FROM t WHERE val = ?";
statement = connection.prepareStatement(sql);
for (int i=0 ; i<10000; i++)
{
  statement.setInt(1, i);
  resultset = statement.executeQuery();
  if (resultset.next())
  {
    pad = resultset.getString("pad");
  }
  resultset.close();
}
statement.close();
```

Oracle JDBC驱动程序提供两个用于支持客户端语句缓存的扩展：隐式和显式语句缓存。如名称所示，前者几乎不需要代码的变更，后者必须显式实现。

通过显式语句缓存，语句依靠Oracle定义的方法来打开和关闭。鉴于这对代码有巨大的影响，并且与隐式语句缓存相比，编写更快的代码会变得更困难。想了解更多信息，请参考*JDBC Developer's Guide*手册。

12

通过隐式语句缓存，当调用close方法时，会将预编译的语句添加到缓存中。然后，当一条新的预编译语句通过prepareStatement方法进行实例化时，就会检查缓存以查明是否拥有相同文本的游标已经存在于其中。

注意　只有实现了java.sql.PreparedStatement和java.sql.CallableStatement接口的类才支持隐式语句缓存。换句话说，普通的语句（基于java.sql.Statement）不支持隐式语句缓存。

下面的代码行展示了在会话级别上启用隐式语句缓存。小心：需要将缓存的大小设置为一个大于0的值。因为这两个方法都是Oracle的扩展程序：

```
((oracle.jdbc.OracleConnection)connection).setImplicitCachingEnabled(true);
((oracle.jdbc.OracleConnection)connection).setStatementCacheSize(50);
```

另一种启用隐式语句缓存的方式是通过OracleDataSource类的setImplicitCachingEnabled和setMaxStatements方法。但是需要注意，setMaxStatements方法已被弃用。

默认情况下，所有预编译的语句都通过隐式语句缓存被缓存起来。当缓存占满时，最近最少使用的那一个就会关闭并被一个新的取代。如果有必要，可以禁用特定语句的缓存。下面的代码举例说明：

```
((oracle.jdbc.OraclePreparedStatement)statement).setDisableStmtCaching(true);
```

本节中作为例子的Java代码摘录自分别实现了测试案例1、2和3的ParsingTest1.java、ParsingTest2.java以及 ParsingTest3.java文件。

12.4.4 ODP.NET

ODP.NET提供对游标生命周期的少量控制。在下面实现测试案例1的代码片段中，ExecuteReader方法在同一时间触发解析、执行和提取调用：

```
sql = "SELECT pad FROM t WHERE val = ";
command = new OracleCommand(sql, connection);
for (int i = 0; i < 10000; i++)
{
  command.CommandText = sql + i;
  reader = command.ExecuteReader();
  if (reader.Read())
  {
    pad = reader[0].ToString();
  }
  reader.Close();
}
```

为避免上面的代码片段执行所产生的全部硬解析，OracleParameter类必须用于传递参数（绑定变量）。下面的代码片段展示了使用它来实现测试案例2。注意，用于查找的值是通过参数定义的，而不是循环传递给sql变量（如上面例子那样）。

```
String sql = "SELECT pad FROM t WHERE val = :val";
OracleCommand command = new OracleCommand(sql, connection);
OracleParameter parameter = new OracleParameter("val", OracleDbType.Int32);
command.Parameters.Add(parameter);
OracleDataReader reader;
```

```
for (int i = 0; i < 10000; i++)
{
  parameter.Value = Convert.ToInt32(i);
  reader = command.ExecuteReader();
  if (reader.Read())
  {
    pad = reader[0].ToString();
  }
  reader.Close();
}
```

使用ODP.NET，不可能实现测试案例3。但是，要达到同样的效果，可以使用客户端语句缓存。有两种方法来启用它并设置缓存的大小。第一种，为所有控制语句缓存并使用特定Oracle home的应用程序，在注册表中设置下面的值。如果设置为0，会禁用语句缓存。如果设置为其他值，则会启用语句缓存，并且这个值指定缓存的大小（<Assembly_Version>是Oracle.DataAccess.dll的完整版本号）。

```
HKEY_LOCAL_MACHINE\SOFTWARE\ORACLE\ODP.NET\<Assembly_Version>\StatementCacheSize
```

第二种方法是直接在代码中通过OracleConnection类提供的Statement Cache Size属性控制语句缓存。基本上，它扮演的角色与注册表是一样的，不过只针对一个单独的连接。下面的代码片段展示了启用语句缓存并将其大小设置为10：

```
String connectString = "User Id=" + user +
                       ";Password=" + password +
                       ";Data Source=" + dataSource +
                       ";Statement Cache Size=10";
OracleConnection connection = new OracleConnection(connectString);
```

注意，在会话级别上的设置会覆盖注册表中的设置。此外，当启用语句缓存时，可以在命令行级别上通过将AddToStatementCache属性设置为false来禁用它。

本节中作为例子的C#代码分别摘录自实现了测试案例1和2的Pars ingTest1.cs和ParsingTest2.cs文件。

12.4.5 PHP

在PHP中，PECL OCI8扩展提供了对游标生命周期的完全控制。例如，在下面实现了测试案例2的代码片段中，请注意显式编码的步骤：

```
$sql = "SELECT pad FROM t WHERE val = :val";
for ($i = 1; $i <= 10000; $i++)
{
  $statement = oci_parse($connection, $sql);
  oci_bind_by_name($statement, ":val", $i, -1, SQLT_INT);
  oci_execute($statement, OCI_NO_AUTO_COMMIT);
  if ($row = oci_fetch_assoc($statement))
  {
        $pad = $row['PAD'];
  }
  oci_free_statement($statement);
}
```

既然可以完全控制游标，也就可以实现测试案例3了。下面的代码片段会举例说明。请注意，为

了避免不必要的软解析，将准备游标（oci_parse和oci_bind_by_name）和关闭游标（oci_free_statement）的函数放置在了循环外面。

```
$sql = "SELECT pad FROM t WHERE val = :val";
$statement = oci_parse($connection, $sql);
oci_bind_by_name($statement, ":val", $i, -1, SQLT_INT);
for ($i = 1; $i <= 10000; $i++)
{
  oci_execute($statement, OCI_NO_AUTO_COMMIT);
  if ($row = oci_fetch_assoc($statement))
  {
    $pad = $row['PAD'];
  }
}
oci_free_statement($statement);
```

PHP不仅可以完全控制游标，而且从OCI8 1.1开始，还支持客户端语句缓存。要使用它，可以使用oci8.statement_cache_size指令。大于0的值会启用客户端语句缓存，并指定缓存游标个数。默认值是20，允许客户端最多缓存20个游标。要更改这个值，请在php.ini配置文件中添加类似以下的内容：

```
oci8.statement_cache_size = 50
```

本节中作为例子使用的PHP代码摘录自ParsingTest1.php、ParsingTest2.php以及 ParsingTest3.php文件。这些文件分别实现了测试案例1、2和3。

12.5　小结

本章描述如何识别、解决以及避开解析问题。核心内容是，通过了解应用程序的工作原理以及利用应用编程接口，从而能够通过在开发阶段编写高效代码来避免解析问题。

鉴于在一个游标的生命周期中，执行阶段紧跟着SQL语句的解析和变量的绑定，有必要了解数据库引擎访问数据时使用的技术。下一章会讨论这方面的内容，并描述如何利用不同类型的索引以及分区方法，以便帮助加速SQL语句的执行。

第13章 优化数据访问

就像第10章描述的那样，执行计划是由多个操作组成的。最常使用的操作是访问、过滤和转换数据。本章主要涉及数据访问操作，也就是，数据库引擎能够访问数据的方式。

基本上在一张表中定位数据的方式仅有两种。第一，是扫描整张表。第二，是基于额外的访问结构（比如索引）或包含表本身（比如散列群集）的结构来进行查找。此外，在分区情况下，会将访问限制到分区的一个子集。这与在本书中寻找特定信息没有区别。要么读完整本书，要么阅读单独一章，或者使用索引或内容表来找出想要的信息。

本章的第一部分将描述，通过看SQL跟踪或动态性能视图提供的运行时统计信息来识别低效的访问路径。第二部分介绍可用的访问方法和使用它们的场合。对于每个访问路径，也会介绍可以用来复制的hint和与它相关的执行计划操作。

注意　本章多个SQL语句包含hint。我这么做不光是要向你展示hint对应的访问路径，同时也举例说明它们的使用。总之，提供的既不是真实的引用也不是完整的语法。可以在*SQL Reference*手册的第2章中找到相关信息。

13.1　识别次优访问路径

第10章介绍了通过查看查询优化器的估值和正确识别的限制，判断执行计划是否高效的方法。重点需要明白即使查询优化器正确选择了最优的执行计划，并不代表这个执行计划就会高效执行。也许在修改了SQL语句或访问结构（例如，增加索引）之后，会想到更好的执行计划。在接下来的几部分中，会介绍用来帮助识别低效访问路径时可以执行的额外检查，以及导致访问路径低效的原因和避免的方法。

13.1.1　识别

最有效的访问路径是能够使用最少的资源来处理数据。因此，要识别访问路径是否高效，可以识别它处理使用的资源数是否可以接受。要做到这些，需要定义如何衡量资源的使用，以及怎样才算是可以接受。此外，还需要考虑检查的可行性。换句话说，也需要考虑执行检查需要做多少工作。它必须尽可能简单。实际上，完善的检查需要花费太多的时间去执行，在实际中这是无法接受的，尤其是

需要处理数十甚至数百条等待优化的SQL语句，或者仅仅是因为你工作的时间很紧。

作为附注，请记得本节关注的是效率，不仅仅是速度。重点需要知道往往最高效的访问路径并不是最快的。正如第15章所述，使用并行处理时，有时即使使用的资源更多，但也可以获得更好的响应时间。当然，当你考虑整个系统时，SQL语句使用越少的资源（换句话说，效率更高），系统的扩展性就会越高，速度越快。这是因为很显然资源是有限的。

作为第一近似值，当访问路径使用的资源数与返回行数（即，返回执行计划里父操作的行数）成正比时，是可接受的。换句话说，当返回少量的行，那么预期的资源使用率会降低，而返回大量行时，资源使用率会升高。因此，检查应该基于返回单行时的资源使用数。

理想情况下，你会衡量数据库引擎使用的全部四种资源类型（CPU、内存、磁盘和网络）的消耗。当然，这可以做到，但不幸的是，获取所有这些指标会花费很多时间和精力，并且通常也只对优化会话中一小部分的SQL语句有效。你也应该考虑当处理一行时，CPU处理时间是依赖处理器的速度的，这在系统与系统之间会有明显的不同。进一步讲，内存使用的总数几乎与返回行数成正比，而磁盘和网络并不是总会用到。实际上，长时间运行的SQL语句使用适度的内存量并且没有磁盘或网络访问也不是罕见的。

幸运的是，有一个数据库度量很容易收集到，它可以告诉你很多数据库引擎工作的信息：逻辑读数，即，在SQL语句执行期间访问的块数。对于它来说有五个好处。第一，逻辑读是CPU-bound操作，因此可以很好地反映CPU使用率。第二，或许逻辑读会导致物理读，因此如果减少逻辑读数，也可能会减少磁盘I/O操作。第三，逻辑读是序列化操作。由于你经常需要优化多用户负载，最小化逻辑读可以很好地避免扩展性问题。第四，在SQL语句和执行计划操作级别上，逻辑读数在SQL跟踪文件和动态性能试图中是现成的。第五，逻辑读数独立于CPU和磁盘I/O子系统的负载。

由于逻辑读数很接近整体的资源消耗数，因此你可以主要处理（至少在第一轮优化中）返回的行中有较高逻辑读数的访问路径。下面是一些通常认为好的“经验法则”。

- 每行小于5个逻辑读的访问路径基本上是好的。
- 每行最多10~15个逻辑读的访问路径基本上可以接受。
- 通常认为每行超过20个逻辑读的访问路径是低效的。换句话说，可能有提升的空间。

要检查每行的逻辑读数，通常有两种方法。第一，利用动态性能视图提供的执行统计信息，然后通过dbms_xplan包来显示（第10章已详细介绍过该技术），下面的执行计划是使用这种方法生成的。对于每个操作，你能看到返回的行数（A-Rows列）和为了返回行执行的逻辑读数（buffer列）：

```
SELECT * FROM t WHERE n1 BETWEEN 6000 AND 7000 AND n2 = 19
----------------------------------------------------------------
| Id  | Operation                   | Name    | A-Rows | Buffers |
----------------------------------------------------------------
|   0 | SELECT STATEMENT            |         |      3 |      28 |
|*  1 |  TABLE ACCESS BY INDEX ROWID| T       |      3 |      28 |
|*  2 |   INDEX RANGE SCAN          | T_N2_I  |     24 |       4 |
----------------------------------------------------------------

  1 - filter(("N1">=6000 AND "N1"<=7000))
  2 - access("N2"=19)
```

第二种方法是利用SQL跟踪提供的信息（第3章已经详细介绍过该技术）。以下代码段引用自

TKPROF生成的输出，使用的是与上一个例子相同的查询。请注意，返回行数（Row列）和逻辑读（cr属性）与之前的指标吻合。

```
Rows     Row Source Operation
-------  ---------------------------------------------------
      3  TABLE ACCESS BY INDEX ROWID T (cr=28 pr=0 pw=0 time=80 us)
     24   INDEX RANGE SCAN T_N2_I (cr=4 pr=0 pw=0 time=25 us)(object id 39684)
```

基于之前提到的经验法则，可以接受这样的执行计划作为例子来使用。实际上，访问路径返回的每行逻辑读数大约是9（28/3）。让我们来看看同样的SQL语句执行计划糟糕时是什么样子。请注意，糟糕是因为访问路径返回的每行逻辑读数是130（390/3），并不是因为它包含全表扫描！

```
--------------------------------------------------------
| Id  | Operation          | Name | A-Rows | Buffers |
--------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      3 |     390 |
|*  1 |  TABLE ACCESS FULL| T    |      3 |     390 |
--------------------------------------------------------

   1 - filter(("N2"=19 AND "N1">=6000 AND "N1"<=7000))
```

需要再次强调本节是关于访问路径的。因此，你必须仅在访问路径层面考虑这些指标，而不是针对整个SQL语句。实际上，在SQL语句级别上这些指标或许会造成误导。要理解可能发生的问题，让我们来检查以下查询。如果是在SQL语句级别（大概是操作0）上，那么执行了387逻辑读来返回一行数据。换句话说，这会导致错误地将其归类为低效的。然而，如果访问操作的指标（操作2）正确列入考虑范围内，那么逻辑读数（387）和返回行数（160）的比率，会将这个访问路径归类为高效的。本例中的问题是操作1对操作2返回的行使用sum函数。结果，它永远都只会返回单行并且“隐藏”了访问路径的性能指标：

```
SELECT sum(n1) FROM t WHERE n2 > 246

---------------------------------------------------------
| Id  | Operation           | Name | A-Rows | Buffers |
---------------------------------------------------------
|   0 | SELECT STATEMENT    |      |      1 |     387 |
|   1 |  SORT AGGREGATE     |      |      1 |     387 |
|*  2 |   TABLE ACCESS FULL| T    |    160 |     387 |
---------------------------------------------------------

   2 - filter("N2">246)
```

如果你真的只能看SQL语句级别的指标（例如，由于SQL跟踪文件不包含执行计划），那么使用之前提供的经验法会变得很困难，仅仅因为你没有足够的信息。然而，在这种情况下，至少对于简单的SQL语句来说，可以尝试猜测访问路径指标而适应经验法则。比如，可以仔细检查SQL语句是否存在聚合，找出SQL语句中引用了多少张表，然后对应引用表的数量，按比例增加经验法则中的限制。

13.1.2 误区

检查逻辑读数时，必须注意两个会曲解指标的误区。第一个与一致读有关，第二个与行预取有关。

1. 一致读

对于每一条SQL语句，数据库引擎都会保证处理数据的一致性。为了达到这个目的，数据块的一致性副本会基于当前数据块和回滚块在运行时创建。要执行这样的操作需要完成数个逻辑读。因此，访问路径操作执行的逻辑读数非常依赖于需要重建的块数。以下代码引用自read_consistency.sql脚本生成的输出。请注意使用的查询与上节相同。根据执行统计信息，会返回相同的行数（实际上返回相同的数据）。然而，它会执行更多的逻辑读（相比28，一共执行了354）。会造成这个影响是因为修改了数据块的另一个会话需要执行该查询。由于在查询开始时修改并未提交，数据库引擎就必须重建这些块。这会导致更高的逻辑读：

```
SELECT * FROM t WHERE n1 BETWEEN 6000 AND 7000 AND n2 = 19

-------------------------------------------------------------------
| Id  | Operation                    | Name    | A-Rows | Buffers |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |      3 |     354 |
|*  1 |  TABLE ACCESS BY INDEX ROWID| T       |      3 |     354 |
|*  2 |   INDEX RANGE SCAN           | T_N2_I  |     24 |     139 |
-------------------------------------------------------------------

   1 - filter(("N1">=6000 AND "N1"<=7000))
   2 - access("N2"=19)
```

2. 行预取

从优化的角度来看，应该避免基于行的处理。例如，当客户端从数据库取回数据时，它可以逐行取回，或者更好些，一次取回多行。这个技术，被称为行预取（row prefetching），会在第15章中详细介绍。现在，让我们只看它对逻辑读数的影响。简单地说，每当数据库引擎访问一个块，逻辑读就会计数一次。针对全表扫描，会有两个极端。如果将行预取设置为1，返回每行大约一个逻辑读。如果将行预取设置为大于每个表块中存储的行数，那么逻辑读就接近表的块数。以下代码引用自row_prefetching.sql脚本生成的输出。在第一个执行中，行预取设置为2（这个值的选择会在15.5节中介绍），逻辑读数（5388）大约是行数（10 000）的一半。在第二个执行中，由于行预取数（100）高于每个块的平均行数（25），逻辑读数（488）大约等于块数（401）：

```
SQL> SELECT num_rows, blocks, round(num_rows/blocks) AS rows_per_block
  2  FROM user_tables
  3  WHERE table_name = 'T';

NUM_ROWS BLOCKS ROWS_PER_BLOCK
-------- ------ --------------
   10000    401             25

SQL> set arraysize 2

SQL> SELECT * FROM t;

-----------------------------------------------------
| Id  | Operation          | Name | A-Rows | Buffers |
-----------------------------------------------------
|   0 | SELECT STATEMENT   |      |  10000 |    5388 |
```

```
|   1 |  TABLE ACCESS FULL| T    |  10000 |    5388 |
-----------------------------------------------------

SQL> set arraysize 100

-----------------------------------------------------
| Id  | Operation         | Name | A-Rows | Buffers |
-----------------------------------------------------
|   0 | SELECT STATEMENT  |      |  10000 |     488 |
|   1 |  TABLE ACCESS FULL| T    |  10000 |     488 |
-----------------------------------------------------
```

注意 在SQL*Plus中，通过arraysize系统变量管理行预取数。默认值是15。

考虑到行预取对逻辑读数的依赖，每当出于测试目的使用诸如SQL*Plus之类的工具执行SQL语句时，都应该仔细将行预取值设置得与应用一致。换句话说，用来测试的工具预取的行数应与应用一致。如果不这样做，会导致很多错误的结果。

当执行的操作被阻塞时（例如，聚合操作），SQL引擎会在内部使用行预取。结果，当聚合是执行计划的一部分时，访问路径的逻辑读数会非常接近块数。换句话说，无论行预取设置成什么，每次SQL引擎访问一个块，它都会包含所有行。下面举例说明：

```
SQL> set arraysize 2

SQL> SELECT sum(n1) FROM t;

------------------------------------------------------
| Id  | Operation          | Name | A-Rows | Buffers |
------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |     388 |
|   1 |  SORT AGGREGATE    |      |      1 |     388 |
|   2 |   TABLE ACCESS FULL| T    |  10000 |     388 |
------------------------------------------------------
```

13.1.3 原因

造成低效访问路径的主要原因有以下几个。

- 没有使用适合的访问结构（比如索引）。
- 使用了适合的访问结构，但是SQL语句的语法不允许查询优化器使用它。
- 表或索引是分区的，但是无法修剪。结果，所有的分区都需要访问。
- 表和/或索引没有适当的分区。

除了之前列表中的例子之外，还有另外两种情况会导致低效访问路径。

- 查询优化器做出错误判断时，可能是由于缺少对象统计信息，或是由于对象统计信息过旧，或由于使用了错误的查询优化器配置。这些没有放在上面的列表中，是因为默认对象统计信息必须是最近的并且查询优化器也是配置正确的（第8章和第9章详细介绍了这两个话题）。
- 当查询优化器自身出现问题时，比如，当出现内部bug或底层限制时。我也不会处理这些，因为bug或查询优化器限制涉及的问题太少了。

13.1.4　解决方案

正如上一部分描述的那样，要高效执行SQL语句，目标就是最小化逻辑读，或者换句话说，使用访问路径访问更少的块。要达到这个目标，或许需要增加新的访问结构（比如，索引）或者改变物理设计（比如，对表或者它们的索引进行分区）。给定的SQL语句，会有很多访问结构和物理设计的组合。幸运的是，为了使选择更容易，可以根据选择性将SQL语句（或者更容易些，数据访问操作）分成以下两大类别：

- 弱选择性操作
- 强选择性操作

选择性很重要，因为访问结构和设计对弱选择性操作支持较好，对强选择性操作支持较差，反之亦然。然而，请注意这两个类别并没有明确的界限。相反，它是依赖于操作的，依赖于它处理的数据和数据存储的方式。例如，数据分布和每块存储行数严重影响着性能。换句话说，并没有这样绝对的说法：选择率小于0.1必定是强选择性，而超过这个值就是弱选择性（或其他任何你想到的值）。尽管如此，实际上限制的范围通常是0.05~0.25。如图13-1所示，你只能确认接近0或1的值。

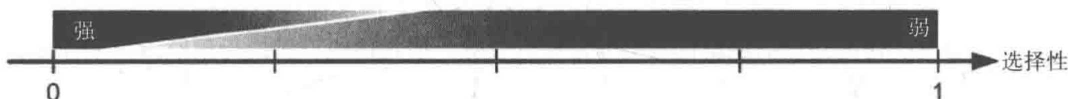


图13-1　在强、弱选择性之间并没有固定的界限

重点需要明白要决定一个操作的类别，与它返回的行数完全无关，只与选择性有关。例如，一个操作返回500 000行与选择的访问路径完全无关。相反，一个操作的选择性为0.001，可以明确地把它放在强选择性类别中。

类别对于选择访问路径的类型很重要，它关联着高效的执行计划。图13-2概括地将选择性与访问路径关联在一起，通常来说这是最优化的方式。当使用合适的索引时，可以高效地执行强选择性操作。在本章稍后的部分，可以看到在一些场景中rowid访问或散列群集也可能会有帮助。另一方面，通过读取全表，可以高效执行弱选择性操作。在这两种可能性之间，分区表和散列群集扮演着重要的角色。



图13-2　指定的访问路径只有在指定的选择性范围内才能高效执行

注意　将数据存储在Exadata存储服务器上时，使用smart scan操作可以利用**存储索引**（storage index）来减少从磁盘物理读取的数据量。因此，一些平衡选择性的操作或者强选择性的操作，可以高效执行读取全表。我们不能控制存储索引，它们由Exadata存储服务器自动管理。因此，本章不会介绍关于存储索引的内容。

让我们来看两个演示实验。在第一个实验中，取回单行数据，而第二个实验取回了上千行数据。

1. 取回单行

这个实验使用access_structures_1.sql脚本，目的是用取回一行数据所需要的逻辑读数与以下适当的访问结构进行对比：

- 带有主键（primary key）的堆表（heap table）
- 索引组织表（index_organized table）
- 主键作为群集键（cluster key）的单表散列群集（single-table hash cluster）

注意　本章只介绍处理SQL语句期间如何利用不同类型的段（比如表、群集和索引）最小化逻辑读。可以在*Oracle Database Concepts*手册中找到它们的基本信息，尤其是"Schema Objects"这一章。

下面是用于实验的查询。请注意，id列是这个表的主键。存在值为6的行，并且rid变量保存着对应行的rowid：

```
SELECT * FROM sales WHERE id = 6

SELECT * FROM sales WHERE rowid = :rid
```

由于逻辑读数与索引高度相关，实验在保存了10、10 000和1 000 000行的表上执行。图13-3汇总了结果。它们阐述了以下四种主要事实。

- 对于所有的访问结构，都是通过rowid（显然，要读取这行保存的块，你无法做到比这个还少的逻辑读）执行单个逻辑读的。
- 对于堆表来说，至少需要两个逻辑读：一个用于索引，另一个用于表。随着行数的增加，索引高度的增加，逻辑读数也会增加。
- 访问索引组织表可以比访问堆表少一个逻辑读。
- 对于单表散列群集，不仅逻辑读数不依赖于行数，而且它总是导致单个逻辑读。

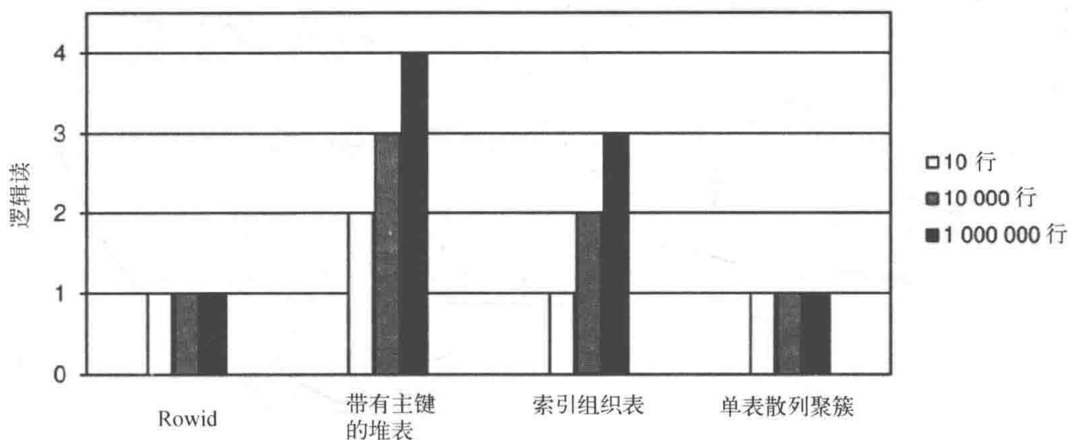


图13-3　不同的访问结构导致不同的逻辑读数

总之，要取回单行，一个"普通"的表加上索引是最低效的访问结构。然而，正如我在本章后续

描述的那样，最常用的是“普通”的表，因为只有在特殊场景才可以利用其他访问结构。

2. 取回多行

这个实验基于access_structures_1000.sql脚本，目的是用取回上千行数据所需的逻辑读数，与以下适当的访问结构进行对比。

- 没有索引的非分区表。
- 列表分区表。Prod_category列是分区列。
- 单表散列群集。Prod_category列是群集键。
- 在prod_category列上有索引的非分区表。对于这个实验，会测试表中的行分布在两个不同的段的情况（因此，存在不同的群集因子）。

测试的数据集包含918 843行。下面的查询显示prod_category列的数据分布情况：

```
SQL> SELECT prod_category, count(*), ratio_to_report(count(*)) over() AS selectivity
  2  FROM sales
  3  GROUP BY prod_category
  4  ORDER BY count(*);

PROD_CATEGORY    COUNT(*) SELECTIVITY
-------------- ---------- -----------
Hardware            15357        .017
Photo               95509        .104
Electronics        116267        .127
Peripherals        286369        .312
Software/Other     405341        .441
```

以下是用来测试的查询：

```
SELECT sum(amount_sold) FROM sales WHERE prod_category = 'Hardware'

SELECT sum(amount_sold) FROM sales WHERE prod_category = 'Photo'

SELECT sum(amount_sold) FROM sales WHERE prod_category = 'Electronics'

SELECT sum(amount_sold) FROM sales WHERE prod_category = 'Peripherals'

SELECT sum(amount_sold) FROM sales WHERE prod_category = 'Software/Other'

SELECT sum(amount_sold) FROM sales
```

对于每一个查询，都会记录逻辑读数。图13-4汇总了结果，产生了以下四个要点。

- 没有索引的非分区表需要的逻辑读数与选择性无关。因此，它只在弱选择性时高效。
- 因为表已经根据prod_category列进行分区，所以列表分区表的单独一个分区需要的逻辑读数与选择性成正比。因此，在所有情况下，会实施最小逻辑读。
- 单表散列群集需要的逻辑读数仅跟选择性中等和高的值成正比（正如稍后会看到的，当选择性强时，散列群集会很有用。然而，在这个实验中，由于数据分布不均匀，它们处于劣势）。
- 通过索引读表需要的逻辑读数非常依赖于数据物理分布。因此，仅知道选择性不足以发现访问路径是否能高效处理数据。

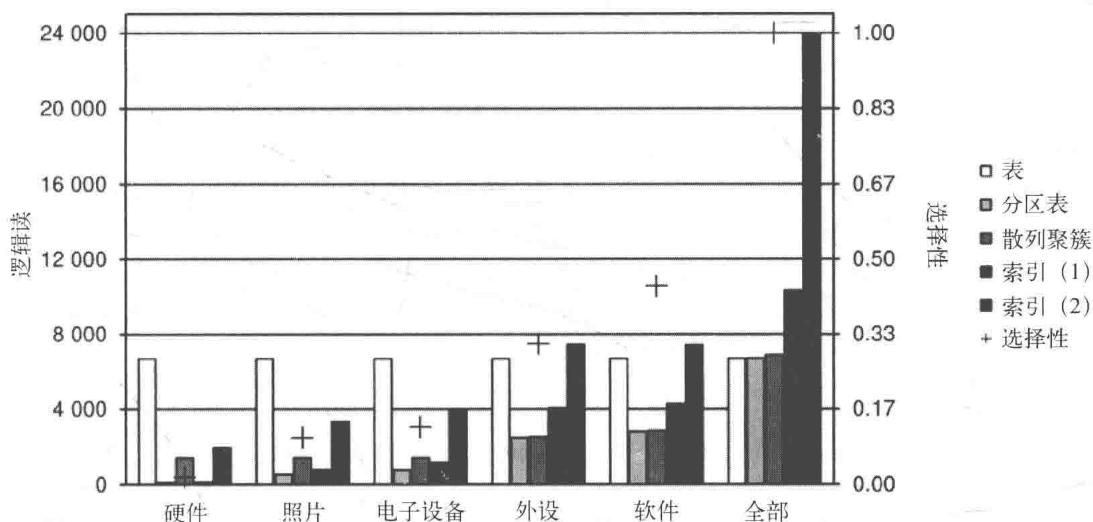


图13-4 特定的访问路径仅对特定范围的选择性高效执行

现在你知道在不同的情况下高效访问数据的主要可行方法，接下来该详细介绍用来处理强和弱选择性SQL语句的访问路径了。

13.2 弱选择性的 SQL 语句

要高效处理数据，弱选择性的SQL语句需要使用全表扫描或全分区扫描。但是在大多数情况下，仅全表扫描可用。这主要有三个原因。第一，分区是企业版的选项。因此，如果使用标准版你将不能使用它，或者你没有分区选项的授权。第二，即使可以使用分区选项，实际上并不是所有的表都会分区。第三，一张表仅被有限数量的列分区。结果就是即使表是分区的，并不是所有的SQL语句都会引用它来利用分区技术，除非它们都引用分区键（partitioning key），这在实际中通常不会发生。

特殊情况下，全表扫描和全分区扫描会被全索引扫描替代。这种情况下，目的不再是利用索引来搜索特定的值，而仅是因为它们要比表小。

13.2.1 全表扫描

所有的堆表上都可以执行全表扫描。由于这种扫描没有任何特殊要求，有时它是唯一可用的访问路径。下面的查询是个例子。请注意在执行计划中，`TABLE ACCESS FULL`操作相当于全表扫描。该例子也展示了如何使用`full` hint来强制执行全表扫描：

```
SELECT /*+ full(t) */ * FROM t WHERE n2 = 19

----------------------------------
| Id  | Operation          | Name |
----------------------------------
|   0 | SELECT STATEMENT   |      |
|*  1 |  TABLE ACCESS FULL| T    |
----------------------------------

  1 - filter("N2"=19)
```

在全表扫描时，服务器进程连续读取表高水位线下所有的块。直到10.2版本（包括该版本）服务器进程才开始执行缓冲区缓存读取。从11.1版本之后，该类磁盘I/O操作依赖于需要读取的块数，目标表的小部分块已在缓冲区缓存中，无论是否将BUFFER_POOL存储参数设置为KEEP。简单地说，当从磁盘读取的块数较低或使用了KEEP缓冲池时，服务器进程就执行缓冲区缓存读取。此外，它们执行直接读。这个选项执行直接读主要是确保大量数据不必通过缓冲区缓存加载（在这里会被立即丢弃）。

全表扫描执行的最小逻辑读数依赖于块数，而不是行数。如果表中包含大量空或者接近空的块，就会导致次优的性能。显然，需要读取块才能知道它是否包含数据。一个导致表产生大量分布稀疏块最常见的场景就是当表删除多于插入时。下面的例子，引用自full_scan_hwm.sql脚本生成的输出。

- 最初，查询执行了468次逻辑读，返回40行：

```
SQL> SELECT * FROM t WHERE n2 = 19;

SQL> SELECT last_output_rows, last_cr_buffer_gets, last_cu_buffer_gets
  2  FROM v$session s, v$sql_plan_statistics p
  3  WHERE s.prev_sql_id = p.sql_id
  4  AND s.prev_child_number = p.child_number
  5  AND s.sid = sys_context('userenv','sid')
  6  AND p.operation_id = 1;

LAST_OUTPUT_ROWS LAST_CR_BUFFER_GETS LAST_CU_BUFFER_GETS
---------------- ------------------- -------------------
              40                 468                   1
```

- 接着，删除几乎所有的行（10 000行中的9960行）。然而，执行查询的逻辑读数并没有改变。换句话说，许多空块被无用地访问了：

```
SQL> DELETE t WHERE n2 <> 19;

9960 rows deleted.

SQL> SELECT * FROM t WHERE n2 = 19;

SQL> SELECT last_output_rows, last_cr_buffer_gets, last_cu_buffer_gets
  2  FROM v$session s, v$sql_plan_statistics p
  3  WHERE s.prev_sql_id = p.sql_id
  4  AND s.prev_child_number = p.child_number
  5  AND s.sid = sys_context('userenv','sid')
  6  AND p.operation_id = 1;

LAST_OUTPUT_ROWS LAST_CR_BUFFER_GETS LAST_CU_BUFFER_GETS
---------------- ------------------- -------------------
              40                 468                   1
```

- 要降低高水位线，需要物理重组表。如果表存储在自动段空间管理的表空间中，那么可以使用下面的 SQL语句。请注意，必须启用行迁移，因为在重组期间，行或许会获得一个新的rowid：

```
SQL> ALTER TABLE t ENABLE ROW MOVEMENT;

SQL> ALTER TABLE t SHRINK SPACE;
```

- 重组后，查询仅执行了23逻辑读，返回40行：

```
SQL> SELECT * FROM t WHERE n2 = 19;

SQL> SELECT last_output_rows, last_cr_buffer_gets, last_cu_buffer_gets
  2  FROM v$session s, v$sql_plan_statistics p
  3  WHERE s.prev_sql_id = p.sql_id
  4  AND s.prev_child_number = p.child_number
  5  AND s.sid = sys_context('userenv','sid')
  6  AND p.operation_id = 1;

LAST_OUTPUT_ROWS LAST_CR_BUFFER_GETS LAST_CU_BUFFER_GETS
---------------- ------------------- -------------------
              40                  23                   0
```

请注意，全表扫描执行的逻辑读数强烈依赖于行预取的设置。关于这方面的例子，请参考13.1.2节。

13.2.2 全分区扫描

当选择性非常弱时（即，接近1），全表扫描是获取数据最有效的方法。随着选择性的降低，全表扫描会访问许多不需要的块。由于使用索引并不益于弱选择性，因此分区是用来减少逻辑读数最常用的选项。使用分区的原因是利用查询优化器的能力去对分区处理中包含的不相关处理数据做排除。这个特性称作分区裁剪（partition pruning）。

要对一个SQL语句使用分区裁剪有两个基本的先决条件。第一，表必须是分区的。第二，必须在SQL语句中指定对分区键的限制或联接条件。如果这两个条件可以满足，那么查询优化器会用一个或多个全分区扫描替换全表扫描。但在实践中，事情没那么简单。实际上，查询优化器需要处理多个特殊场景或许也可能不会导致分区裁剪。要更好地理解这些场景，接下来的部分会详述分区裁剪的基础，也包含高级裁剪技术比如OR（where条件里使用or）、multicolumn（多列）、subquery（子查询）和join-filter pruning（联接过滤裁剪）。之后是一些关于如何实施分区的实用性建议。请注意索引分区会在本章稍后的13.3节介绍。

13.2.3 范围分区

要举例说明分区裁剪的工作原理，让我们来检查基于pruning_range.sql脚本的多个例子。Test表就是范围分区并且使用以下SQL语句创建。为了能够展示所有类型的分区裁剪，分区键由两列组成：n1和d1。表由n1列四个不同的值和基于d1列的月份进行分区。这代表每年有48个分区：

```
CREATE TABLE t (
  id NUMBER,
  d1 DATE,
  n1 NUMBER,
  n2 NUMBER,
  n3 NUMBER,
  pad VARCHAR2(4000),
  CONSTRAINT t_pk PRIMARY KEY (id)
)
PARTITION BY RANGE (n1, d1) (
  PARTITION t_1_jan_2014 VALUES LESS THAN (1, to_date('2014-02-01','yyyy-mm-dd')),
```

```
  PARTITION t_1_feb_2014 VALUES LESS THAN (1, to_date('2014-03-01','yyyy-mm-dd')),
  PARTITION t_1_mar_2014 VALUES LESS THAN (1, to_date('2014-04-01','yyyy-mm-dd')),
  ...
  PARTITION t_4_oct_2014 VALUES LESS THAN (4, to_date('2014-11-01','yyyy-mm-dd')),
  PARTITION t_4_nov_2014 VALUES LESS THAN (4, to_date('2014-12-01','yyyy-mm-dd')),
  PARTITION t_4_dec_2014 VALUES LESS THAN (4, to_date('2015-01-01','yyyy-mm-dd'))
)
```

警告　像这样存在两个分区键的案例，如果第一个键无法唯一定位一个单独分区，数据引擎会仅使用第二个键来插入新行。因此，当指定PARTITION BY RANGE子句时，n1列会指定在d1列前。

图13-5是Test表的图示。

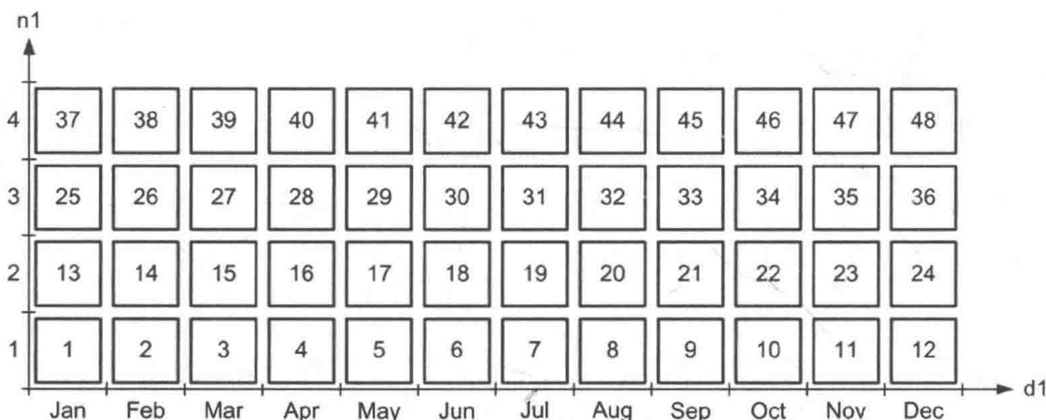


图13-5　Test表每年由48个分区组成[①]

每个分区都可以通过其名称或在表中的"位置"（后者在图13-5中）进行识别。当然，两个值之间的映射可以在数据字典中找到。下面的查询展示了如何从user_tab_partitions视图中获取这些信息：

```
SQL> SELECT partition_name, partition_position
  2  FROM user_tab_partitions
  3  WHERE table_name = 'T'
  4  ORDER BY partition_position;

PARTITION_NAME PARTITION_POSITION
-------------- ------------------
T_1_JAN_2014                    1
T_1_FEB_2014                    2
T_1_MAR_2014                    3
...
T_4_OCT_2014                   46
T_4_NOV_2014                   47
T_4_DEC_2014                   48
```

① 自12.1.0.2起，PARTITON RANGE ITERATOR操作也用于基于区域图的分区裁剪。

对于这个表来说，如果在分区键上有限制，查询优化器就能识别出并且能够排除包含处理不相关数据的分区。由于数据字典中包含分区的界限，因此查询优化器可以拿它们与SQL语句中的限制或联接条件作对比。然而，由于限制，分区裁剪并不总是可用。下一小节会展示不同的例子来指出查询优化器在何时、如何使用分区裁剪。

注意 这部分例子中只使用了查询。这并不代表分区裁剪只能应用于查询。实际上，它也可以应用于同样的SQL语句，如UPDATE和DELETE。只是为了方便起见我才只使用查询。

1. PARTITON RANGE SINGLE

下面的SQL语句，WHERE子句包含两个限制：对应分区键的每列。在这样的情况下，查询优化器识别出只有单独一个分区包含相关数据。结果，在执行计划里会显示PARTITION RANGE SINGLE操作。重点需要知道它的子操作（TABLE ACCESS FULL）并不是对整张表进行全表扫描。相反，只访问了单独一个分区。这也被Starts列的值所证实。Pstart和Pstop列指明了访问的分区：

```
SELECT * FROM t WHERE n1 = 3 AND d1 = to_date('2014-07-19','yyyy-mm-dd')

---------------------------------------------------------------
| Id  | Operation              | Name | Starts | Pstart| Pstop|
---------------------------------------------------------------
|   0 | SELECT STATEMENT       |      |      1 |       |      |
|   1 |  PARTITION RANGE SINGLE|      |      1 |    31 |   31 |
|*  2 |   TABLE ACCESS FULL    | T    |      1 |    31 |   31 |
---------------------------------------------------------------

  2 - filter("D1"=TO_DATE(' 2014-07-19 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND "N1"=3)
```

图13-6为这种行为的示意图。

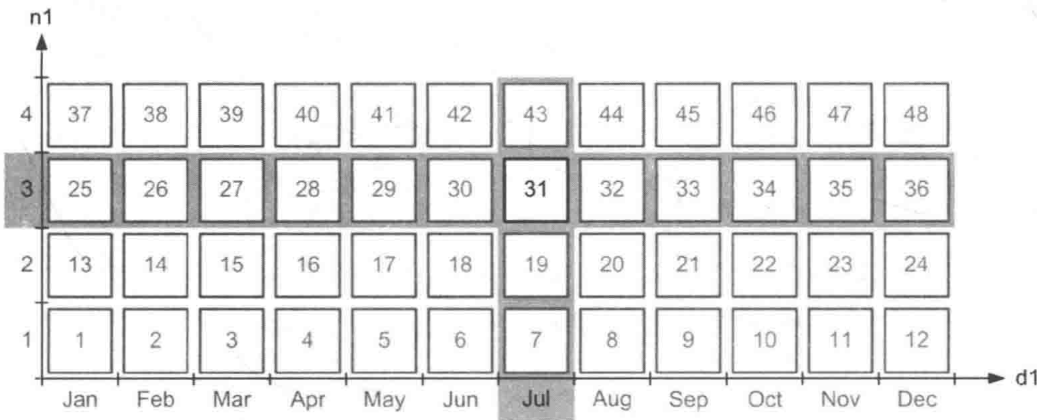


图13-6 PARTITION RANGE SINGLE操作的表现

正如下面查询的输出显示，Pstart和Pstop列的分区数与user_tab_partitions视图中的partition_position列的值吻合：

```
SQL> SELECT partition_name
  2  FROM user_tab_partitions
  3  WHERE table_name = 'T'
  4  AND partition_position = 31;

PARTITION_NAME
--------------
T_3_JUL_2014
```

每当绑定变量在限制中使用时，查询优化器就不能在解析阶段判断该访问哪个分区。这种情况下，分区裁剪在运行时执行。执行计划不会改变，但是Pstart和Pstop列会设置成KEY。这代表发生了分区裁剪，但是在解析阶段，查询优化器并不知道哪个分区包含了相关数据：

```
SELECT * FROM t WHERE n1 = :n1 AND d1 = to_date(:d1,'YYYY-MM-DD')

----------------------------------------------------------------
| Id  | Operation               | Name | Starts | Pstart| Pstop |
----------------------------------------------------------------
|   0 | SELECT STATEMENT        |      |      1 |       |       |
|   1 |  PARTITION RANGE SINGLE|      |      1 |   KEY |   KEY |
|*  2 |   TABLE ACCESS FULL     | T    |      1 |   KEY |   KEY |
----------------------------------------------------------------

  2 - filter(("D1"=TO_DATE(:D1,'YYYY-MM-DD') AND "N1"=:N1))
```

2. PARTITION RANGE ITERATOR

上一部分介绍的执行计划包含PARTITION RANGE SINGEL操作。这是因为查询优化器识别出用户只有单独一个分区包含相关处理数据。显然，会存在需要访问多个分区的情况。例如，在下面的查询中，限制使用了小于条件（<）而不是相等条件（=），因此，操作变成了PARTITION RANGE ITERATOR，并且Pstart和Pstop列显示被访问的分区范围（请查看图13-7）。此外，Starts列显示操作1只执行了一次，但是操作2每个分区执行了一次。换句话说，执行了多次全分区扫描：

```
SELECT * FROM t WHERE n1 = 3 AND d1 < to_date('2014-07-19','YYYY-MM-DD')

------------------------------------------------------------------
| Id  | Operation                 | Name | Starts | Pstart| Pstop |
------------------------------------------------------------------
|   0 | SELECT STATEMENT          |      |      1 |       |       |
|   1 |  PARTITION RANGE ITERATOR|      |      1 |    25 |    31 |
|*  2 |   TABLE ACCESS FULL       | T    |      7 |    25 |    31 |
------------------------------------------------------------------

  2 - filter(("N1"=3 AND "D1"<TO_DATE(' 2014-07-19 00:00:00', 'syyyy-mm-dd hh24:mi:ss')))
```

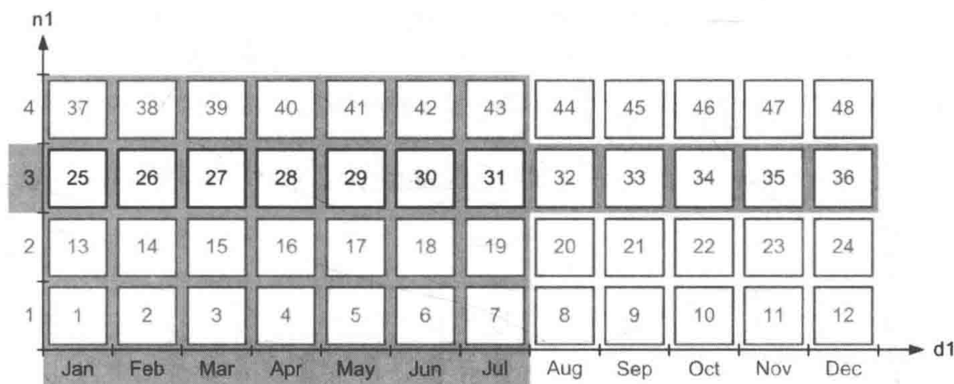


图13-7 PARTITION RANGE ITERATOR操作表现[①]

PARTITION RANGE ITERATOR操作也会用于当限制基于分区键的前导列时。下面的查询举例说明，限制应用于分区键的第一个列上。请注意也会访问分区37。这是因为如果d1列存在大于2014年12月31日的值，那么n1列等于3的行就存储在这个分区里：

```
SELECT * FROM t WHERE n1 = 3

-------------------------------------------------------------------
| Id  | Operation                | Name | Starts | Pstart| Pstop |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT         |      |      1 |       |       |
|   1 |  PARTITION RANGE ITERATOR|      |      1 |    25 |    37 |
|*  2 |   TABLE ACCESS FULL      | T    |     13 |    25 |    37 |
-------------------------------------------------------------------

   2 - filter("N1"=3)
```

就像这个操作的名称所暗示的那样，它只对连续的分区范围有效。当使用非连续分区时，就会用到下一节的操作。

3. PARTITION RANGE INLIST

如果限制基于一个或多个由超过一个元素组成的IN条件时，那么在执行计划中就会有PARTITION RANGE INLIST操作。使用这个操作，Pstart和Pstop列不会给出访问哪个分区的具体信息。相反，它们会显示KEY(I)的值。这代表对IN条件中的每个值分别执行分区裁剪。此外，Starts列显示访问了多少个分区（本例为2）：

```
SELECT * FROM t WHERE n1 IN (1,3) AND d1 = to_date('2014-07-19','YYYY-MM-DD')

-----------------------------------------------------------------
| Id  | Operation              | Name | Starts | Pstart| Pstop |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT       |      |      1 |       |       |
|   1 |  PARTITION RANGE INLIST|      |      1 |KEY(I) |KEY(I) |
```

① 自12.1.0.2起，PARTITION RANGE ITERATOR操作也用于基于区域图的分区裁剪。

```
|* 2 |   TABLE ACCESS FULL    | T    |      2 |KEY(I) |KEY(I) |
-------------------------------------------------------------

  2 - filter(("D1"=TO_DATE(' 2014-07-19 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND
              INTERNAL_FUNCTION("N1")))
```

在此特定案例中，根据WHERE子句，可以推断出仅会访问分区7和31。图13-8说明了这一点。

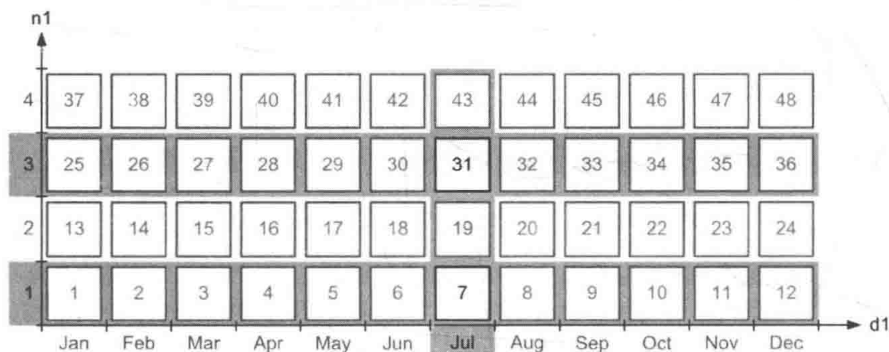


图13-8　PARTITION RANGE INLIST操作表现

当然，如果IN条件中的值是完全分散的，那就有可能大部分分区都要访问到。这种情况下执行计划会认为所有分区都需要访问，就会使用下一节的操作。

4. PARTITION RANGE ALL

如果在分区键上没有限制，那么所有分区都必须访问。这种情况下，执行计划包含PARTITION RANGE ALL操作，并且Starts、Pstart和Pstop列显示所有分区都会被访问：

```
SELECT * FROM t WHERE n3 BETWEEN 6000 AND 7000

-----------------------------------------------------------
| Id  | Operation            | Name | Starts | Pstart| Pstop |
-----------------------------------------------------------
|   0 | SELECT STATEMENT     |      |      1 |       |       |
|   1 |  PARTITION RANGE ALL |      |      1 |     1 |    48 |
|*  2 |   TABLE ACCESS FULL  | T    |     48 |     1 |    48 |
-----------------------------------------------------------

  2 - filter(("N3">=6000 AND "N3"<=7000))
```

在主键上使用不等式作为限制时，也会使用这个相同的执行计划。下面的查询是一个例子：

```
SELECT * FROM t WHERE n1 != 3 AND d1 != to_date('2014-07-19','YYYY-MM-DD')
```

当在分区键上基于表达式或函数进行限制时，也会使用相同的执行计划。例如，下面的查询，在n1列上加1，并且通过to_char函数修改d1列：

```
SELECT * FROM t WHERE n1 + 1 = 4 AND to_char(d1,'YYYY-MM-DD') = '2014-07-19'
```

这代表要利用分区裁剪，不仅要基于分区键的限制，而且不能在分区键上使用表达式或函数。如果必须要使用表达式，从11.1版本开始，可以选择虚拟列作为主键。

5. PARTITION RANGE EMPTY

当查询优化器发现没有分区保存相关处理数据时，执行计划中会出现这个特别的操作PARTITION RANGE EMPTY。例如，下面的查询查找的数据没有分区保存（对于n1列来说，值5超出了范围）。同样需要注意的是，不仅会将Pstart和Pstop列设置为INVALID，而且只会执行操作1（不会消耗任何资源，因为基本上这是个空操作）：

```
SELECT * FROM t WHERE n1 = 5 AND d1 = to_date('2014-07-19','YYYY-MM-DD')

-----------------------------------------------------------------
| Id  | Operation              | Name | Starts | Pstart| Pstop |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT       |      |      1 |       |       |
|   1 |  PARTITION RANGE EMPTY|      |      1 |INVALID|INVALID|
|*  2 |   TABLE ACCESS FULL    | T    |      0 |INVALID|INVALID|
-----------------------------------------------------------------

   2 - filter(("D1"=TO_DATE(' 2014-07-19 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND "N1"=5))
```

6. PARTITION RANGE OR

本节介绍的裁剪类型，也叫作OR裁剪，它是分区键上的分隔谓词用在了WHERE子句上（由OR条件组合的谓词）。下面的查询就是这样的例子。当使用这种类型的裁剪时，执行计划中出现PARTITION RANGE OR操作。请注意Pstart和Pstop列也会被设置为KEY(OR)。在下面的例子中，根据Starts列，会访问18个分区。之所以是18个分区，是因为尽管应用在n1列上的限制访问分区25和37，但是应用在d1列上的限制访问分区1、3、13、15、25、27、37和39（分区1用来找出是否存在包含n1列值在PARTITION BY RANGE子句中未指定上限的行）：

```
SELECT * FROM t WHERE n1 = 3 OR d1 = to_date('2014-03-06','YYYY-MM-DD')

-----------------------------------------------------------
| Id  | Operation          | Name | Starts | Pstart| Pstop |
-----------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |       |       |
|   1 |  PARTITION RANGE OR|      |      1 |KEY(OR)|KEY(OR)|
|*  2 |   TABLE ACCESS FULL| T    |     18 |KEY(OR)|KEY(OR)|
-----------------------------------------------------------

   2 - filter(("N1"=3 OR "D1"=TO_DATE(' 2014-03-06 00:00:00', 'syyyy-mm-dd hh24:mi:ss')))
```

7. PARTITION RANGE SUBQUERY

在之前的部分中，所有用来分区裁剪的限制都是基于文字或绑定变量。然而，限制是联接条件的情况也很常见。每当基于分区键联接时，不仅查询优化器不会总利用分区裁剪，而且在一些情况下这么做也是不明智的。要选择最低成本的执行计划，查询优化器需要在三种策略中选择。

注意 第14章会详细介绍联接方法。

第一种策略是避免使用分区裁剪。下面的查询（请注意tx表是t表的副本；唯一的不同是tx表没有分区）举例说明，t表上没有执行分区裁剪。实际上，因为操作4是PARTITON RANGE ALL，操作5处理

了所有分区。在本例中，执行计划是非常低效的。尤其是当查询的选择性强时：

```
SELECT * FROM tx, t WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND tx.id = 19

-----------------------------------------------------------------------
| Id  | Operation                    | Name   | Starts | Pstart| Pstop |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT             |        |      1 |       |       |
|*  1 |  HASH JOIN                   |        |      1 |       |       |
|   2 |   TABLE ACCESS BY INDEX ROWID| TX     |      1 |       |       |
|*  3 |    INDEX UNIQUE SCAN         | TX_PK  |      1 |       |       |
|   4 |   PARTITION RANGE ALL        |        |      1 |     1 |    48 |
|   5 |    TABLE ACCESS FULL         | T      |     48 |     1 |    48 |
-----------------------------------------------------------------------

  1 - access("TX"."D1"="T"."D1" AND "TX"."N1"="T"."N1")
  3 - access("TX"."ID"=19)
```

这种策略总是有效的。然而，如果联接条件的选择性不接近于1，或者换句话说，在应用分区裁剪的情景中，就会导致糟糕的性能。

第二种策略是使用NESTED LOOPS操作来执行联接并访问表，这会触发作为第二个子操作的分区裁剪。实际上，正如在第10章中讨论的那样，NESTED LOOPS操作是关联合并操作，因此，它的第一个子操作控制着第二个子操作。下面的例子展示了这样的场景。请注意PARTITION RANGE ITERATOR操作以及Pstart和Pstop列的值证实发生了分区裁剪。根据Starts列，会访问单独一个分区。在本例中，下面的执行计划要远比第一种策略使用的执行计划高效：

```
SELECT * FROM tx, t WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND tx.id = 19
-----------------------------------------------------------------------
| Id  | Operation                    | Name   | Starts | Pstart| Pstop |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT             |        |      1 |       |       |
|   1 |  NESTED LOOPS                |        |      1 |       |       |
|   2 |   TABLE ACCESS BY INDEX ROWID| TX     |      1 |       |       |
|*  3 |    INDEX UNIQUE SCAN         | TX_PK  |      1 |       |       |
|   4 |   PARTITION RANGE ITERATOR   |        |      1 |   KEY |   KEY |
|*  5 |    TABLE ACCESS FULL         | T      |      1 |   KEY |   KEY |
-----------------------------------------------------------------------

  3 - access("TX"."ID"=19)
  5 - filter(("TX"."D1"="T"."D1" AND "TX"."N1"="T"."N1"))
```

这种策略只有在NESTED LOOP操作（本例为操作2）的第一个子操作返回的行数很小时才能表现良好。否则，很可能同样的分区会被第二个子操作（本例为操作4）访问很多次。

第三种策略是使用HASH JOIN或MERGE JOIN操作来执行联接。没有基于常规分区使用这些联接方法的联接条件进行裁剪。实际上，正如第10章介绍的那样，它们是非关联合并操作，因此，两个子操作会单独执行。这种情况下，查询优化器会利用另一种类型的分区裁剪，子查询裁剪（subquery pruning）。它的目的是通过递归查询找出第二个子操作应该访问哪个分区。为了这个目的，SQL引擎执行递归查询（通过第一个子操作访问表）来取回联接条件与第二个子操作分区键匹配的列。接着，查询存储在数据字典中第二个子操作的分区定义，识别出被第二个子操作访问的分区，这样就仅需扫描它们。下

面的查询举例说明。请注意PARTITION RANGE SUBQUERY操作和Pstart与Pstop列的值（KEY(SQ)）证实发生了分区裁剪。根据Starts列，访问了单独一个分区：

```
SELECT * FROM tx, t WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND tx.id = 19

------------------------------------------------------------------------
| Id  | Operation                     | Name  | Starts | Pstart| Pstop |
------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |       |      1 |       |       |
|*  1 |  HASH JOIN                    |       |      1 |       |       |
|   2 |   TABLE ACCESS BY INDEX ROWID| TX    |      1 |       |       |
|*  3 |    INDEX UNIQUE SCAN          | TX_PK |      1 |       |       |
|   4 |   PARTITION RANGE SUBQUERY    |       |      1 |KEY(SQ)|KEY(SQ)|
|   5 |    TABLE ACCESS FULL          | T     |      1 |KEY(SQ)|KEY(SQ)|
------------------------------------------------------------------------

  1 - access("TX"."D1"="T"."D1" AND "TX"."N1"="T"."N1")
  3 - access("TX"."ID"=19)
```

事实上，SQL引擎递归执行了以下操作来找出需要访问的分区。这个递归查询取回包含tbloridx$part$num函数与相关数据的分区数。操作5可以利用这个信息使用分区裁剪。例如，本例中只需扫描分区37：

```
SQL> SELECT DISTINCT TBL$OR$IDX$PART$NUM("T", 0, 1, 0, "N1", "D1") AS PART_NUM
  2  FROM (SELECT "TX"."N1" "N1", "TX"."D1" "D1"
  3        FROM "TX" "TX"
  4        WHERE "TX"."ID"=19)
  5  ORDER BY 1;

  PART_NUM
----------
        37
```

很明显，仅在递归查询执行引起的开销小于分区裁剪增加的开销时，使用第三种技术才有意义。对于本例中使用的查询，第二种和第三种策略生成的执行计划效率是非常相似的。然而，如果选择性弱，第三种策略生成的执行计划会更有效率。

8. PARTITION RANGE JOIN-FILTER

子查询裁剪是非常有用的优化技术。然而，正如前面部分讨论的那样，部分SQL语句会执行两次。为了避免这种两次执行的情况，从11.1版本开始，数据引擎提供了另一类的分区裁剪：联接过滤裁剪（join-filter pruning，也被称为bloom-filter pruning）。要理解它的工作原理，让我们看一下与子查询裁剪部分相同的查询生成的执行计划。请注意会出现一些新东西：PART JOIN FILTER CREATE操作、PARTITION RANGE JOIN-FILTER操作和在Name，Pstart和Pstop列上的字符串BF0000。

```
SELECT * FROM tx, t WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND tx.id = 19

-----------------------------------------------------------------------------------
| Id  | Operation                     | Name    | Starts | E-Rows | Pstart| Pstop |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |         |      1 |        |       |       |
|*  1 |  HASH JOIN                    |         |      1 |      7 |       |       |
|   2 |   PART JOIN FILTER CREATE     | :BF0000 |      1 |      1 |       |       |
|   3 |    TABLE ACCESS BY INDEX ROWID| TX      |      1 |      1 |       |       |
```

```
|* 4 |      INDEX UNIQUE SCAN         | TX_PK  |     1 |     1 |         |         |
|  5 |   PARTITION RANGE JOIN-FILTER  |        |     1 | 10000 | :BF0000 | :BF0000 |
|  6 |    TABLE ACCESS FULL           | T      |     1 | 10000 | :BF0000 | :BF0000 |
-----------------------------------------------------------------------------------

  1 - access("TX"."N1"="T"."N1" AND "TX"."D1"="T"."D1")
  4 - access("TX"."ID"=19)
```

执行计划的执行如下所示。

- 操作3和4通过tx_pk索引访问tx表。
- 根据操作3返回的数据，操作2基于在联接条件（tx.d1和tx.n1）中使用的列值创建内存结构（布隆过滤器）。
- 根据操作2创建的内存结构，操作5能够利用分区裁剪，因此能够只访问包含相关数据的分区。这种情况下，会访问单独分区（请查看Starts列）。

9. PARTITION RANGE MULTI-COLUMN

如果分区键由多列组合而成，重点需要观察当限制没有固定在所有列上时会发生什么。主要问题是，查询优化器会利用分区裁剪吗？答案是，会利用多列裁剪（multicolumn pruning）。多列裁剪的目的很简单：不依赖于限制定义的列，总会发生分区裁剪。

让我们看一下这个特性在之前相同实验中的作用。由于Test表的分区键是由两列组成的，因此需要考虑两种情况：限制会应用在第一列或第二列。下面的查询举例说明前者：

```
SELECT * FROM t WHERE n1 = 3

---------------------------------------------------------------
| Id | Operation                | Name | Starts | Pstart| Pstop |
---------------------------------------------------------------
|  0 | SELECT STATEMENT         |      |      1 |       |       |
|  1 |  PARTITION RANGE ITERATOR|      |      1 |    25 |    37 |
|* 2 |   TABLE ACCESS FULL      | T    |     13 |    25 |    37 |
---------------------------------------------------------------

  2 - filter("N1"=3)
```

下面的查询举例说明后者。请注意PARTITION RANGE MULTI-COLUMN操作以及Pstart和Pstop列的值证实发生了分区裁剪；然而，并没有提供具体访问了哪个分区：

```
SELECT * FROM t WHERE d1 = to_date('2014-07-19','YYYY-MM-DD')

-------------------------------------------------------------------
| Id | Operation                    | Name | Starts | Pstart| Pstop |
-------------------------------------------------------------------
|  0 | SELECT STATEMENT             |      |      1 |       |       |
|  1 |  PARTITION RANGE MULTI-COLUMN|      |      1 |KEY(MC)|KEY(MC)|
|* 2 |   TABLE ACCESS FULL          | T    |      8 |KEY(MC)|KEY(MC)|
-------------------------------------------------------------------

  2 - filter("D1"=TO_DATE(' 2014-07-19 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
```

10. PARTITION RANGE AND

在某些情况下，正如前面部分介绍的那样，查询优化器可以利用多个裁剪技术。例如，请查看下

面的查询：

```
SELECT * FROM tx, t WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND t.n1 = 3 AND tx.n2 = 42
```

查询优化器需要对这样的SQL语句考虑至少使用以下两种裁剪技术。

❑ 基于t.n1 = 3限制的分区裁剪：

```
-----------------------------------------------------------------------------
| Id  | Operation                 | Name | Starts | Pstart| Pstop | Buffers |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT          |      |      1 |       |       |     889 |
|*  1 |  HASH JOIN                |      |      1 |       |       |     889 |
|*  2 |   TABLE ACCESS FULL       | TX   |      1 |       |       |     403 |
|   3 |   PARTITION RANGE ITERATOR|      |      1 |    25 |    37 |     486 |
|*  4 |    TABLE ACCESS FULL      | T    |     13 |    25 |    37 |     486 |
-----------------------------------------------------------------------------

  1 - access("TX"."N1"="T"."N1" AND "TX"."D1"="T"."D1")
  2 - filter(("TX"."N2"=42 AND "TX"."N1"=3))
  4 - filter("T"."N1"=3)
```

❑ 基于tx.d1 = t.d1和tx.n1 = t.n1联接条件（利用tx.n2 = 42限制）的分区裁剪：

```
-----------------------------------------------------------------------------------
| Id  | Operation                   | Name    | Starts | Pstart| Pstop | Buffers |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |         |      1 |       |       |     963 |
|*  1 |  HASH JOIN                  |         |      1 |       |       |     963 |
|   2 |   PART JOIN FILTER CREATE   | :BF0000 |      1 |       |       |     403 |
|*  3 |     TABLE ACCESS FULL       | TX      |      1 |       |       |     403 |
|   4 |   PARTITION RANGE JOIN-FILTER|        |      1 |:BF0000|:BF0000|     560 |
|*  5 |    TABLE ACCESS FULL        | T       |     15 |:BF0000|:BF0000|     560 |
-----------------------------------------------------------------------------------

  1 - access("TX"."N1"="T"."N1" AND "TX"."D1"="T"."D1")
  3 - filter("TX"."N2"=42)
  5 - filter("T"."N1"=3)
```

从11.2版本起，查询优化器可以同时利用多个裁剪技术。这可以保证访问最少的分区（对比这三种情况的Starts列）。下面的例子显示当使用这种叫作AND裁剪的类型时执行计划中会出现PARTITION RANGE AND操作。也请注意Pstart和Pstop列被设置为KEY(AP)。在本例中，分区裁剪基于限制（t.n1 = 3）和联接条件（tx.d1 = t.d1 AND tx.n1 = t.n1）。请注意为联接条件创建的布隆过滤器：

```
SELECT * FROM tx, t WHERE tx.d1 = t.d1 AND tx.n1 = t.n1 AND t.n1 = 3 AND tx.n2 = 42
```

```
---------------------------------------------------------------------------------
| Id  | Operation                 | Name    | Starts | Pstart| Pstop | Buffers |
---------------------------------------------------------------------------------
|   0 | SELECT STATEMENT          |         |      1 |       |       |     630 |
|*  1 |  HASH JOIN                |         |      1 |       |       |     630 |
|   2 |   PART JOIN FILTER CREATE | :BF0000 |      1 |       |       |     403 |
|*  3 |    TABLE ACCESS FULL      | TX      |      1 |       |       |     403 |
|   4 |   PARTITION RANGE AND     |         |      1 |KEY(AP)|KEY(AP)|     227 |
|*  5 |    TABLE ACCESS FULL      | T       |      6 |KEY(AP)|KEY(AP)|     227 |
---------------------------------------------------------------------------------
```

```
1 - access("TX"."N1"="T"."N1" AND "TX"."D1"="T"."D1")
3 - filter(("TX"."N2"=42 AND "TX"."N1"=3))
5 - filter("T"."N1"=3)
```

13.2.4　散列和列表分区

上一部分只介绍了范围分区。散列和列表分区也可以使用范围分区介绍的大部分技术。

下面是散列分区可用的技术。Pruning_hash.sql脚本提供的执行计划例子中包含了这些操作：

- PARTITION HASH SINGLE
- PARTITION HASH ITERATOR
- PARTITION HASH INLIST
- PARTITION HASH ALL
- PARTITION HASH SUBQUERY
- PARTITION HASH JOIN-FILTER
- PARTITION HASH AND

下面是列表分区可用的技术。Pruning_list.sql脚本提供的执行计划例子中包含了这些操作：

- PARTITION LIST SINGLE
- PARTITION LIST ITERATOR
- PARTITION LIST INLIST
- PARTITION LIST ALL
- PARTITION LIST EMPTY
- PARTITION LIST OR
- PARTITION LIST SUBQUERY
- PARTITION LIST JOIN-FILTER
- PARTITION LIST AND

13.2.5　复合分区

关于复合分区没什么可介绍的。基本上，在分区级别应用的一切也适用于子分区。不过，至少举个例子来说明。下面的Test表是根据范围（range）进行分区（基于d1列），而子分区是根据列表（list）进行分区（基于n1列）。下面的SQL语句引用自pruning_composite.sql脚本，用来创建该表。请注意本例中，也是每年48个分区：

```
CREATE TABLE t (
  id NUMBER,
  d1 DATE,
  n1 NUMBER,
  n2 NUMBER,
  n3 NUMBER,
  pad VARCHAR2(4000),
  CONSTRAINT t_pk PRIMARY KEY (id)
)
```

```
PARTITION BY RANGE (d1)
SUBPARTITION BY LIST (n1)
SUBPARTITION TEMPLATE (
  SUBPARTITION sp_1 VALUES (1),
  SUBPARTITION sp_2 VALUES (2),
  SUBPARTITION sp_3 VALUES (3),
  SUBPARTITION sp_4 VALUES (4)
)(
  PARTITION t_jan_2014 VALUES LESS THAN (to_date('2014-02-01','YYYY-MM-DD')),
  PARTITION t_feb_2014 VALUES LESS THAN (to_date('2014-03-01','YYYY-MM-DD')),
  PARTITION t_mar_2014 VALUES LESS THAN (to_date('2014-04-01','YYYY-MM-DD')),
  PARTITION t_apr_2014 VALUES LESS THAN (to_date('2014-05-01','YYYY-MM-DD')),
  PARTITION t_may_2014 VALUES LESS THAN (to_date('2014-06-01','YYYY-MM-DD')),
  PARTITION t_jun_2014 VALUES LESS THAN (to_date('2014-07-01','YYYY-MM-DD')),
  PARTITION t_jul_2014 VALUES LESS THAN (to_date('2014-08-01','YYYY-MM-DD')),
  PARTITION t_aug_2014 VALUES LESS THAN (to_date('2014-09-01','YYYY-MM-DD')),
  PARTITION t_sep_2014 VALUES LESS THAN (to_date('2014-10-01','YYYY-MM-DD')),
  PARTITION t_oct_2014 VALUES LESS THAN (to_date('2014-11-01','YYYY-MM-DD')),
  PARTITION t_nov_2014 VALUES LESS THAN (to_date('2014-12-01','YYYY-MM-DD')),
  PARTITION t_dec_2014 VALUES LESS THAN (to_date('2015-01-01','YYYY-MM-DD'))
)
```

图13-9是Test表的示意图。如果将其与之前的（请看图13-5）进行对比，贯穿整张表唯一的不同就是没有任何值标记子分区的位置。实际上，子分区的位置基于它的“父”分区。

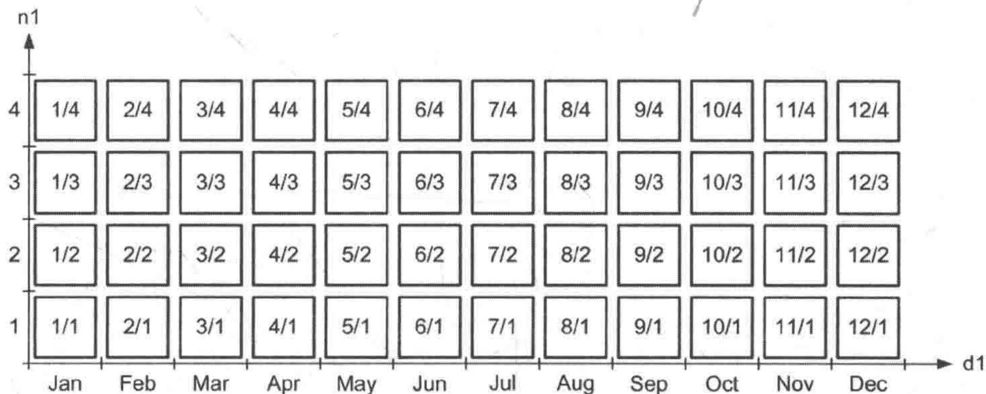


图13-9 Test表由每年48个分区组成

当然，本例中name和position之间的映射也可以在数据字典中找到。下面的查询展示了如何在user_tab_partitions和user_tab_subpartitions视图中获取这些信息：

```
SQL> SELECT subpartition_name, partition_position, subpartition_position
  2  FROM user_tab_partitions p, user_tab_subpartitions s
  3  WHERE p.table_name = 'T'
  4  AND s.table_name = p.table_name
  5  AND s.partition_name = p.partition_name
  6  ORDER BY p.partition_position, s.subpartition_position;

SUBPARTITION_NAME PARTITION_POSITION SUBPARTITION_POSITION
```

```
---------------- ----------------- --------------------
T_JAN_2014_SP_1                  1                    1
T_JAN_2014_SP_2                  1                    2
T_JAN_2014_SP_3                  1                    3
T_JAN_2014_SP_4                  1                    4
T_FEB_2014_SP_1                  2                    1
...
T_NOV_2014_SP_4                 11                    4
T_DEC_2014_SP_1                 12                    1
T_DEC_2014_SP_2                 12                    2
T_DEC_2014_SP_3                 12                    3
T_DEC_2014_SP_4                 12                    4
```

下面的查询是在分区和子分区级别上都应用限制的例子。操作在上一部分都介绍过。操作1应用在分区级别，而操作2应用在子分区级别。在分区级别，访问分区1和7。对于每个分区，只访问子分区3。图13-10展示了这个行为。请注意Pstart和Pstop列的值与之前查询数据字典返回值的对应关系：

```
SELECT * FROM t WHERE n1 = 3 AND d1 < to_date('2014-07-19','YYYY-MM-DD')

-----------------------------------------------------------------
| Id  | Operation                | Name | Starts | Pstart| Pstop |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT         |      |      1 |       |       |
|   1 |  PARTITION RANGE ITERATOR|      |      1 |     1 |     7 |
|   2 |   PARTITION LIST SINGLE  |      |      7 |     3 |     3 |
|*  3 |    TABLE ACCESS FULL     | T    |      7 |   KEY |   KEY |
-----------------------------------------------------------------

  3 - filter("D1"<TO_DATE(' 2014-07-19 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
```

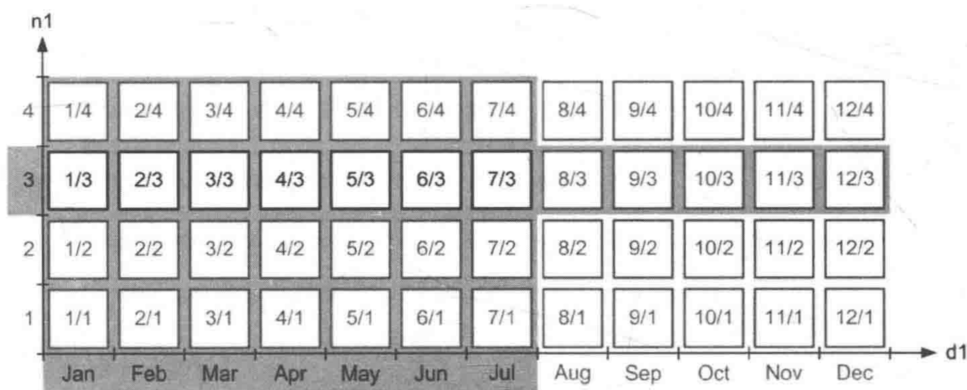


图13-10　复合分区裁剪的表现

13.2.6　设计要素

正如上节介绍的那样，查询优化器可以在大部分情况下使用分区裁剪。表13-1总结了对于每种类型的分区方法何时会发生分区裁剪以及最常见的SQL条件。

表13-1 导致分区裁剪的条件*

条 件	范 围	列 表	散 列
Equality (=)	✓	✓	✓
IN	✓	✓	✓
BETWEEN、>、>=、<或<=	✓	✓	
IS NULL	✓	✓	

* 不等值（例如!=或<>）、NOT IN、IS NOT NULL条件和基于表达式和函数的限制不会发生分区裁剪。

需要设计分区表时，选择分区键和分区方法大概是需要做出的最重要的决定。目的是高效地利用分区裁剪处理尽可能多的SQL语句。例如，如果SQL语句频繁根据天来处理数据，那就应该按照天来分区。或者，如果SQL语句频繁地按照国家来处理数据，那就应该按照国家来分区。如果错误地分区，就无法利用分区裁剪。下面的四个特点是你的应用需要仔细考虑的，因为它们最影响分区策略。

(1) 哪个列是限制会应用的，以及它的频率。

(2) 这里保存了什么类型的数据。

(3) 限制会使用什么SQL条件。

(4) 数据是否会定期压缩或删除以及处理的标准。

第一和第四条是选择分区键的关键。第二和第三条用于选择分区方法。让我们详细讨论一下。

重点是要知道限制会应用到哪些列上，因为如果在分区键上没有限制就不会使用分区裁剪。换句话说，基于这个标准，应选择在有限的列数内应用限制。实际上，在不同的SQL语句上应用多个不同的限制很常见。因此，也需要知道不同SQL语句使用的频率。这样，就可以决定哪个限制是最需要优化的。总之，只需要考虑有弱选择性的限制。实际上，强选择性的限制可以使用其他访问结构（比如，索引）优化。

一旦知道了可能作为分区键的列，就该查看它们存储的数据了。目的是找出应该应用哪种分区方法。为此，需要重点注意两件事。第一，只有范围和列表分区允许集合“相关”数据（例如，七月的全部销售额或所有欧洲国家）或者准确地映射特定的值与特定的分区。第二，每种分区方法仅适用于特定类型的数据。

- Range适用于自然连续的值。典型实例是时间戳和序列生成的数字。
- List适用于常见且有限的值。典型实例是所有类型的状态信息（例如，enabled、disabled、processed）和描述人（例如，性别、婚姻状况）或事（例如， 国家、邮编、货币、类别、格式）的属性。
- Hash适用于不同的值超出分区数很多的所有类型数据（例如，客户编号）。

分区方法需要适用于这种数据，也适用于限制应用组成分区键的列。在表13-1描述的与散列（hash）分区有关的限制也应该考虑到。此外，注意，即便技术上能够在列表（list）分区表上使用范围条件，这种不能使用分区裁剪的情况也很常见。实际上，列表分区表中的数据是自然排列的，而不是连续的。

若发生定期压缩或删除，也应该考虑它们是否可以使用分区。比如，删除或截断一个分区要比删除它包含的数据快得多。这种策略通常只用在范围或列表分区上。

一旦选择了分区键，就需要确定分区键是否可以修改。这样的修改代表着移动行到另一个分区（与

删除旧行然后在另一个分区里重新插入的操作非常相似），即改变它的rowid。通常，行从来不改变它的rowid。因此，如果发生改变，不光要启用表级别的行迁移来允许数据库引擎做出修改，同时使用rowid的应用也需要特殊处理。

最后一个注释，但在我看来也是非常重要的注释，用来组织一些最常见的错误，这些错误我已经在实施分区的项目中经历过了。错误是在设计与实现数据库与应用时并没有使用分区，而是在之后才分区。通常，这种方法注定会失败。我强烈建议在项目最初就计划使用分区。如果你认为自己可以在之后轻松地使用它，那么就要体验到“大惊喜”了。

13.2.7　全索引扫描

数据库引擎不仅可以利用索引提取rowid列表，也可以将它们作为指针来读取表中对应的行，但它也可以直接读取索引键部分的列值，从而避免根据rowid再去访问表。多亏了这项重要的优化技术，当索引包含查询需要访问的所有数据时，全表扫描或全分区扫描会由*全索引扫描*（full index scan）替代。并且由于索引段通常要比表段小，用来减小逻辑读会很有用。

全索引扫描主要用于三种情况。第一是当索引存储查询需要的所有列时。比如，因为n1列有索引，下面的查询可以使用全索引扫描。下面的执行计划证实了没有执行访问表的操作（请注意本节所有的例子都基于index_full_scan.sql脚本）：

```
SELECT /*+ index(t t_n1_i) */ n1 FROM t WHERE n1 IS NOT NULL

-----------------------------------
| Id  | Operation        | Name   |
-----------------------------------
|   0 | SELECT STATEMENT |        |
|*  1 |  INDEX FULL SCAN | T_N1_I |
-----------------------------------

  1 - filter("N1" IS NOT NULL)
```

重点需要明白这个执行计划是合理的，因为WHERE子句（n1 IS NOT NULL）的条件确定了索引存储着所有需要处理的数据（在n1列上的NOT NULL约束也能有同样的效果，因为它确保没有NULL值插入）。否则，因为单列B树索引不保存NULL值，就必须执行全表扫描了。

正如上面例子所示，INDEX FULL SCAN操作可以由index hint强制使用，根据它的结构扫描索引。这么做的优点是可以通过索引键取回存储的数据。缺点是如果索引块没在缓冲区缓存中的话，那么它会使用单块读从数据文件中读取它（跟随叶子块中的一个指针到下一个/上一个叶子块）。由于全索引扫描会读取许多数据，所以这通常是低效的。要改善这种情况下的性能，可以使用*索引快速全扫描*（index fast full scans）。下面的执行计划举例说明：

```
SELECT /*+ index_ffs(t t_n1_i) */ n1 FROM t WHERE n1 IS NOT NULL
----------------------------------------
| Id  | Operation             | Name   |
----------------------------------------
|   0 | SELECT STATEMENT      |        |
|*  1 |  INDEX FAST FULL SCAN | T_N1_I |
----------------------------------------
```

```
  1 - filter("N1" IS NOT NULL)
```

INDEX FAST FULL SCAN操作可以通过index_ffs hint强制执行，它的特性是使用多块读从数据文件中读取索引块，就像全表扫描访问表一样。在扫描期间，可以简单地丢弃根和分支块，因为所有数据都保存在叶子块中（通常根和分支块只是索引段的一小部分，所以即使读取它们，开销通常也忽略不计）。结果，并不考虑访问的索引结构，因此取回的数据并没有根据索引键来保存。

第二个案例与第一个类似。唯一的不同是数据需要按照索引中存储的顺序传输。例如，正如下面的查询所示，因为指定了ORDER BY 子句，情况就是这样的。由于顺序的关系，仅INDEX FULL SCAN操作可以用来排序操作：

```
SELECT /*+ index(t t_n1_i) */ n1 FROM t WHERE n1 IS NOT NULL ORDER BY n1

-----------------------------------
| Id  | Operation        | Name   |
-----------------------------------
|   0 | SELECT STATEMENT |        |
|*  1 |  INDEX FULL SCAN | T_N1_I |
-----------------------------------

  1 - filter("N1" IS NOT NULL)
```

由于INDEX FULL SCAN操作执行的磁盘I/O操作要比INDEX FAST FULL SCAN操作低效，前者只在排序时才会用到。

警告 nls_sort参数影响ORDER BY操作。如果未将它的值设置为binary，那么仅在索引列的数据类型不受NLS设置（比如，NUMBER和DATE）影响或使用语言索引（linguistic index，13.3.2节提供了关于这种索引的信息）时，索引全扫描才可以用来优化ORDER BY。

默认情况下，索引扫描升序执行。因此，index hint会使查询优化器也按照这种方式执行。要明确指定扫描顺序，可以使用index_asc和index_desc hint。下面的查询介绍了操作方式。执行计划中显示了降序（DESCENDING）扫描：

```
SELECT /*+ index_desc(t t_n1_i) */ n1 FROM t WHERE n1 IS NOT NULL ORDER BY n1 DESC

----------------------------------------------
| Id  | Operation                   | Name   |
----------------------------------------------
|   0 | SELECT STATEMENT            |        |
|*  1 |  INDEX FULL SCAN DESCENDING| T_N1_I |
----------------------------------------------

  1 - filter("N1" IS NOT NULL)
```

第三个案例与count函数有关。如果查询包含它，查询优化器会尝试利用索引而避免全表扫描。下面的查询举例说明。请注意，SORT AGGREGATE操作是用来执行count函数的：

```
SELECT /*+ index_ffs(t t_n1_i) */ count(n1) FROM t
```

```
---------------------------------------
| Id | Operation              | Name   |
---------------------------------------
|  0 | SELECT STATEMENT       |        |
|  1 |  SORT AGGREGATE        |        |
|  2 |   INDEX FAST FULL SCAN| T_N1_I |
---------------------------------------
```

当count处理可为空（nullable）的列时，查询优化器会挑出任何包含该列（因为NULL值无法计数）的索引。当执行count (*)或针对非空列执行count时，查询优化器能够选择任何包含非空列的B树索引（因为只有在本例中索引项的数目保证与行数一致），或任意位图索引。因此，它会考虑选择更小的索引。

即使本节的例子都是基于B树索引，大多数技术也可以用于位图索引。只有两点不同。第一，位图索引不能降序扫描（由于实现的限制）。第二，位图索引总是保存NULL值。因此，它们应用的范围要比B树索引更大。以下查询展示的例子与之前的相似：

```
SELECT /*+ index(t t_n2_i) */ n2 FROM t WHERE n2 IS NOT NULL

---------------------------------------------
| Id  | Operation                    | Name   |
---------------------------------------------
|   0 | SELECT STATEMENT             |        |
|   1 |  BITMAP CONVERSION TO ROWIDS|        |
|*  2 |   BITMAP INDEX FULL SCAN     | T_N2_I |
---------------------------------------------

  2 - filter("N2" IS NOT NULL)

SELECT /*+ index_ffs(t t_n2_i) */ n2 FROM t WHERE n2 IS NOT NULL

----------------------------------------------
| Id  | Operation                     | Name   |
----------------------------------------------
|   0 | SELECT STATEMENT              |        |
|   1 |  BITMAP CONVERSION TO ROWIDS  |        |
|*  2 |   BITMAP INDEX FAST FULL SCAN| T_N2_I |
----------------------------------------------

  2 - filter("N2" IS NOT NULL)

SELECT /*+ index(t t_n2_i) */ n2 FROM t WHERE n2 IS NOT NULL ORDER BY n2

---------------------------------------------
| Id  | Operation                    | Name   |
---------------------------------------------
|   0 | SELECT STATEMENT             |        |
|   1 |  BITMAP CONVERSION TO ROWIDS|        |
|*  2 |   BITMAP INDEX FULL SCAN     | T_N2_I |
---------------------------------------------

  2 - filter("N2" IS NOT NULL)
```

```
SELECT /*+ index_ffs(t t_n2_i) */ count(n2) FROM t

-------------------------------------------------
| Id  | Operation                      | Name    |
-------------------------------------------------
|   0 | SELECT STATEMENT               |         |
|   1 |  SORT AGGREGATE                |         |
|   2 |   BITMAP CONVERSION TO ROWIDS  |         |
|   3 |    BITMAP INDEX FAST FULL SCAN| T_N2_I  |
-------------------------------------------------
```

13.3 强选择性的 SQL 语句

要高效处理强选择性的SQL语句，数据需要通过rowid、索引或单表散列群集访问[1]。这三种会在接下来的部分中介绍。

13.3.1 Rowid 访问

访问一行数据最高效的方法就是在WHERE子句中直接指定它的rowid。然而，要利用这个访问路径，就必须首先获得rowid，保存它，然后在以后的访问中重用它。换句话说，这个方法只有在一行数据至少被访问两次时才会考虑用到。实际上，当SQL语句有强选择性时，这会发生得很频繁。例如，应用使用手动方式维护数据（换句话说，并不是批量）通常会访问同样的行至少两次，至少显示当前数据一次和至少再次保存修改一次。这种情况下，高效利用rowid访问就有意义了。

Oracle其中一个工具（SQL Developer）提供了一个很好的例子。比如，当SQL Developer显示数据时，它会获取数据和它的rowid。比如scott模式下的emp表，工具会执行以下查询。请注意，在SELECT子句中的第一个列就是rowid：

```
SELECT ROWID,"EMPNO","ENAME","JOB","MGR","HIREDATE","SAL","COMM","DEPTNO" FROM "SCOTT"."EMP"
```

稍后，rowid可以用来直接访问特定的行。例如，如果你打开Single Record View对话框（请查看图13-11），修改comm列，并且提交修改，会执行下面的SQL语句。正如你看到的，工具会使用rowid引用修改的行而不是主键（因此节省了从主键索引读取多个块的开销）：

```
UPDATE "SCOTT"."EMP" SET COMM=:sqldevvalue WHERE ROWID = :sqldevgridrowid
```

从优化的角度考虑，使用rowid非常有益，因为可以直接访问行，不需要其他的访问结构（比如索引）。下面的执行计划与这样的SQL语句有关：

```
---------------------------------------------
| Id  | Operation                   | Name |
---------------------------------------------
|   0 | UPDATE STATEMENT            |      |
|   1 |  UPDATE                     | EMP  |
|   2 |   TABLE ACCESS BY USER ROWID| EMP  |
---------------------------------------------
```

[1] 实际上，还有多表散列群集和索引群集。这里不介绍是因为在实际中它们很少使用。

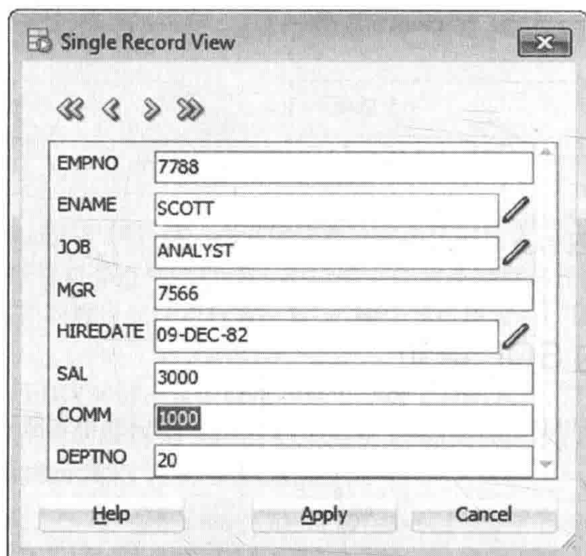


图13-11　SQL Developer的对话框用来显示、浏览和修改数据

请注意，TABLE ACCESS BY USER ROWID操作专门用在当rowid作为参数或文字直接传递时。在下一节，你会看到当rowid从索引中提取出来，会使用TABLE ACCESS BY INDEX ROWID操作替代。访问这些表的效率是一样的；这两个操作仅用来区分rowid的来源。

当一条SQL语句通过IN条件指定多个rowid时，执行计划里会多出额外的INLIST ITERATOR操作。下面的查询举例说明。请注意操作1（父操作）指出了操作2（子操作）被处理了多次。根据操作2的Starts列值，它处理了两次。换句话说，emp表通过rowid访问了两次：

```
SELECT * FROM emp WHERE rowid IN ('AAADGZAAEAAAAAoAAH', 'AAADGZAAEAAAAAoAAI')

-----------------------------------------------------
| Id  | Operation                  | Name | Starts |
-----------------------------------------------------
|   0 | SELECT STATEMENT           |      |      1 |
|   1 |  INLIST ITERATOR           |      |      1 |
|   2 |   TABLE ACCESS BY USER ROWID| EMP  |      2 |
-----------------------------------------------------
```

总之，每当指定行被访问至少两次，就应该考虑在第一次访问时获取rowid，然后利用它来进行后续的访问。

13.3.2　索引访问

索引访问是目前对强选择性SQL语句最常用的访问路径。要使用它们，必须在WHERE子句中使用至少一个限制或通过索引使用联接条件。要这么做，不仅索引列要提供强选择性，同时还要理解通过索引哪种类型的条件才可以高效使用。数据库引擎支持不同类型的索引。在详细介绍B树索引和位图索引支持的访问路径和属性前，重点需要说一下群集因子，或者换句话说，为什么数据分布影响索引扫描的性能（请查看图13-4）。

注意 尽管数据库引擎针对负载的数据（如PDF文档或图片）支持域索引，但在本书中不作介绍。更多的信息请参考Oracle数据库官方文档。

1. 群集因子

正如第8章中介绍的那样，群集因子指示有多少相邻的索引键不引用表中相同的数据块（位图索引是例外，因为总是会将其群集因子设置为索引中键的数量）。下面一张记忆图像或许会有帮助：如果整张表通过索引访问并且在缓冲区缓存中有单独一个缓冲区保存着数据块，那么群集因子是对表执行物理读的数量。例如，图13-12展示索引的群集因子是10（请注意，总共有12行，且仅有2个高亮显示的相邻索引键引用相同的数据块）。

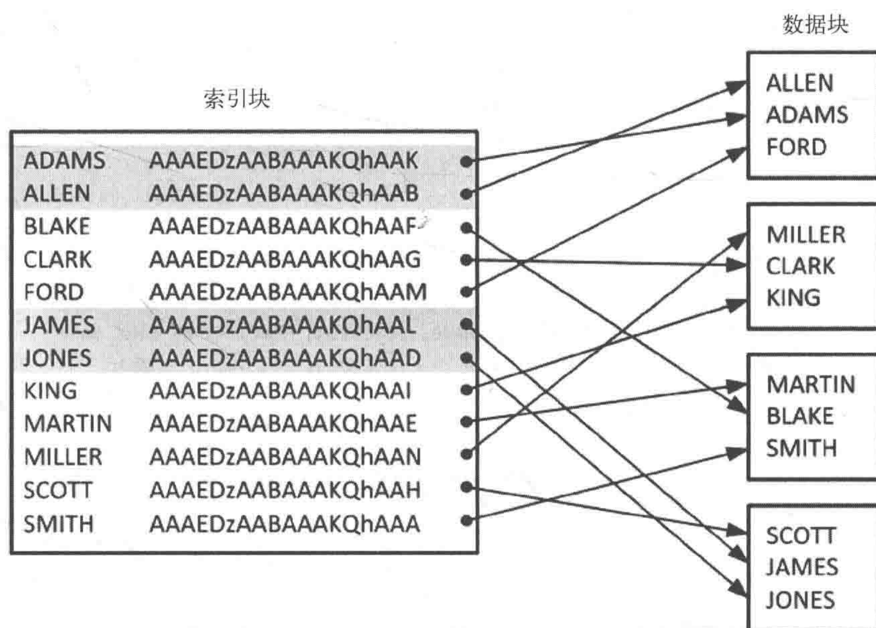


图13-12 索引块与数据块之间的关系

下面的PL/SQL函数可以在clustering_factor.sql脚本中找到，举例说明了它是如何计算的。请注意，这个函数仅能使用在单行B树索引上：

```
CREATE OR REPLACE FUNCTION clustering_factor (
  p_owner IN VARCHAR2,
  p_table_name IN VARCHAR2,
  p_column_name IN VARCHAR2
) RETURN NUMBER IS
  l_cursor             SYS_REFCURSOR;
  l_clustering_factor BINARY_INTEGER := 0;
  l_block_nr           BINARY_INTEGER := 0;
  l_previous_block_nr BINARY_INTEGER := 0;
  l_file_nr            BINARY_INTEGER := 0;
```

```
  l_previous_file_nr  BINARY_INTEGER := 0;
BEGIN
  OPEN l_cursor FOR
    'SELECT dbms_rowid.rowid_block_number(rowid) block_nr, '||
    '       dbms_rowid.rowid_to_absolute_fno(rowid, '''||
                                            p_owner||''','''||
                                            p_table_name||''') file_nr '||
    'FROM '||p_owner||'.'||p_table_name||' '||
    'WHERE '||p_column_name||' IS NOT NULL '||
    'ORDER BY ' || p_column_name ||', rowid';
  LOOP
    FETCH l_cursor INTO l_block_nr, l_file_nr;
    EXIT WHEN l_cursor%NOTFOUND;
    IF (l_previous_block_nr <> l_block_nr OR l_previous_file_nr <> l_file_nr)
    THEN
      l_clustering_factor := l_clustering_factor + 1;
    END IF;
    l_previous_block_nr := l_block_nr;
    l_previous_file_nr := l_file_nr;
  END LOOP;
  CLOSE l_cursor;
  RETURN l_clustering_factor;
END;
```

请注意，它生成的值与数据字典中统计信息的匹配：

```
SQL> SELECT i.index_name, i.clustering_factor,
  2         clustering_factor(user, i.table_name, ic.column_name) AS my_clus_fact
  3  FROM user_indexes i, user_ind_columns ic
  4  WHERE i.table_name = 'T'
  5  AND i.index_name = ic.index_name
  6  ORDER BY i.index_name;

INDEX_NAME CLUSTERING_FACTOR MY_CLUS_FACT
---------- ----------------- ------------
T_PK                     990          990
T_VAL_I                   77           77
```

群集因子计算是悲观的

dbms_stats包使用的算法用来计算群集因子是相当悲观的。

实际上，算法考虑在索引范围扫描期间，只有块被缓存中的前索引键引用到。实际上，早期的多个块可能留在缓存中。结果，群集因子无法准确描述真实的数据分布并不罕见。

为了防止或者解决关于悲观计算的问题，从11.2.0.4版本开始（或者安装了解决bug 13262857的补丁之后）可以通过dbms_stats包来设置table_cached_blocks。它的值从1至255，用来指定会有多少个块期望被缓存。默认值是1。即使建议值总是很难给出，但好像任何大于1的值都会使群集因子更可靠。实际上，默认值实在是太悲观了。此外，dbms_stats.auto_table_cached_blocks的值指定使用255、表块的1%或缓冲区缓存的0.1%，会选择它们之中最小的那个。

从性能的观点看，应该避免基于行的处理。这在本章之前介绍过，多亏了行预取，数据库引擎也会尽可能地避免它。实际上，当必须从同样的块中取出多行时，在单个访问中所有的行都会提取出来而不是每取出一行都会访问一次块（即，逻辑读）。要强调这一点，让我们来看一个基于clustering_factor.sql脚本的例子。下面是由脚本创建的Test表：

```
SQL> CREATE TABLE t (
  2    id NUMBER,
  3    val NUMBER,
  4    pad VARCHAR2(4000),
  5    CONSTRAINT t_pk PRIMARY KEY (id)
  6  );

SQL> INSERT INTO t
  2  SELECT rownum, dbms_random.value, dbms_random.string('p',500)
  3  FROM dual
  4  CONNECT BY level <= 1000;
```

由于id列的值是按照递增顺序插入的，索引支持主键的群集因子接近表中的块数。换句话说，这是好事：

```
SQL> SELECT blocks, num_rows
  2  FROM user_tables
  3  WHERE table_name = 'T';

    BLOCKS   NUM_ROWS
---------- ----------
        80       1000

SQL> SELECT blevel, leaf_blocks, clustering_factor
  2  FROM user_indexes
  3  WHERE index_name = 'T_PK';

    BLEVEL LEAF_BLOCKS CLUSTERING_FACTOR
---------- ----------- -----------------
         1           2                77
```

当整张表通过主键访问时（使用hint强制执行这样的执行计划），查看执行的逻辑读数会很有帮助。第一个实验将行预取设置为2。因此，数据库引擎至少执行500次调用来取回1000行。可以通过查看逻辑读数（Buffers列）来证实这种行为：索引上503，表上539（1042−503）。多亏了群集因子，基本上对于每次调用，都会从索引块上提取两个rowid，并且几乎每次它们的数据也会在相同的数据块上找到：

```
SQL> set arraysize 2

SQL> SELECT /*+ index(t t_pk) */ * FROM t;

---------------------------------------------------------------------------
| Id  | Operation                   | Name | Starts | A-Rows | Buffers |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |      1 |   1000 |    1042 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |      1 |   1000 |    1042 |
|   2 |   INDEX FULL SCAN           | T_PK |      1 |   1000 |     503 |
---------------------------------------------------------------------------
```

第二个实验将行预取设置为100。因此，数据库引擎执行10次调用就足以取回所有行。这个行为也可以通过逻辑读数来证实：索引上是13，表上是87（100–13）。基本上，对于每次调用，100个rowid从索引块中提取出来，而它们表中相应的行通常也会在相同的块中找到，并且只需要访问一次：

```
SQL> set arraysize 100

SQL> SELECT /*+ index(t t_pk) */ * FROM t;

-----------------------------------------------------------------------
| Id  | Operation                   | Name | Starts | A-Rows | Buffers |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |      1 |   1000 |     100 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |      1 |   1000 |     100 |
|   2 |   INDEX FULL SCAN           | T_PK |      1 |   1000 |      13 |
-----------------------------------------------------------------------
```

现在让我们使用更高的群集因子执行相同的实验。要完成这个实验，会将基于随机值的ORDER BY子句添加到INSERT语句中，用来将数据存入表中。唯一改变的统计信息就是群集因子。请注意，这个值接近行数。换句话说，这不是好事：

```
SQL> TRUNCATE TABLE t;

SQL> INSERT INTO t
  2  SELECT rownum, dbms_random.value, dbms_random.string('p',500)
  3  FROM dual
  4  CONNECT BY level <= 1000
  5  ORDER BY dbms_random.value;

SQL> SELECT blocks, num_rows
  2  FROM user_tables
  3  WHERE table_name = 'T';

    BLOCKS   NUM_ROWS
---------- ----------
        80       1000

SQL> SELECT blevel, leaf_blocks, clustering_factor
  2  FROM user_indexes
  3  WHERE index_name = 'T_PK';

    BLEVEL LEAF_BLOCKS CLUSTERING_FACTOR
---------- ----------- -----------------
         1           2               990
```

下面是两个实验的逻辑读数。一方面，两个实验中索引上的逻辑读数都没有改变。因为索引上以完全相同的顺序存储着相同的键（只有对应的rowid不同）。另一方面，两个实验在表上的逻辑读数都接近行数。因为索引上临近的rowid几乎不会涉及相同的块：

```
SQL> set arraysize 2

SQL> SELECT /*+ index(t t_pk) */ * FROM t;
```

```
------------------------------------------------------------------------
| Id  | Operation                   | Name | Starts | A-Rows | Buffers |
------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |      1 |   1000 |    1499 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |      1 |   1000 |    1499 |
|   2 |   INDEX FULL SCAN           | T_PK |      1 |   1000 |     502 |
------------------------------------------------------------------------

SQL> set arraysize 100

------------------------------------------------------------------------
| Id  | Operation                   | Name | Starts | A-Rows | Buffers |
------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |      1 |   1000 |    1003 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |      1 |   1000 |    1003 |
|   2 |   INDEX FULL SCAN           | T_PK |      1 |   1000 |      13 |
------------------------------------------------------------------------
```

总之，使用高群集因子的行预取更低效，因此会执行更高的逻辑读数。群集因子对资源消耗的影响非常高，以至于查询优化器（在第9章中介绍过，尤其是公式9-4）使用群集因子计算索引访问相关的成本，这通常是计算的主要因素。

2. B树索引与位图索引

简单来说，有些情况下只会使用B树索引。如果不是这些情况，大多数情况下也会使用位图索引。表13-2总结了决定使用B树索引和位图索引时需要考虑的特性。

表13-2　仅支持B树索引和位图索引的重要特性

特　　性	B树索引	位图索引
主键和唯一键	✓	
行级锁	✓	
多个索引的高效组合		✓
分区表上的全局索引和非分区索引	✓	

注意　位图索引仅在企业版中可用。

位图索引的使用主要有两种情况限制。第一，只有B树索引可以使用主键和唯一键。这通常没有选择。第二，只有B树索引支持行级锁，因为索引（B树索引和位图索引）中的锁是在内部设置索引项。因为单个位图索引项可能指向数千行，位图索引列的修改可能会阻止其他几千行（引用自相同索引项）的修改，这会在很大程度上抑制可扩展性。另一个位图索引的劣势是，当修改它们时，数据库引擎会生成更多的redo。这是因为，大多数情况下位图索引键要比B树索引键大。

请注意索引的选择性或不重复的键数与选择B树索引或位图索引无关。尽管如此，许多关于位图索引的书和文章还是包含以下建议：

> 位图索引适用于低基数不常修改的数据。当一列不重复值的数量与整个行数相比低很多时，数据具有低基数。
>
> 通过压缩技术，这些索引可以使用最小I/O生成许多rowid。当使用的查询中包含以下特

征时，位图索引可以提供特别有用的访问路径：

- WHERE子句中的多个条件
- 在低基数列上的AND和OR操作
- COUNT函数
- 为空值选择的谓词

——*Oracle Database SQL Tuning Gude* 12*c* Release 1

老实说从我的观点来看，这样的信息至少有些误导。事实上，弱选择性的SQL语句不可能通过从索引获取rowid列表来高效执行。这是因为对于B树索引和位图索引来说，用来建立rowid列表的逻辑读的数量要比使用它们访问表的逻辑读的数量少得多。因此，无论是B树索引还是位图索引，大部分时间都用来读表了。这就是说，位图索引要比具有较少数量的非重复键的B树索引表现得更好（请注意在摘录中，术语基数（cardinality）与本书中使用的有不同的含义）。但是请注意，更好并不代表高效。例如，一个糟糕的产品并不好，因为它仅仅比一个非常糟的产品更好。组合位图索引或许非常高效，但是建立rowid列表仅是开始。之后仍然需要访问行。我也应该提到OR。如果你能想到，就会意识到使用OR组合多个非选择性条件只会带来更弱的选择性，而如果想高效使用索引，那么目标应该是提高选择性。

提示 需要在B树索引和位图索引中选择时，请忽略选择性和基数。

除了表13-2总结的不同之外，当处理相同的SQL条件时，B树索引和位图索引并未显示相同的效率。实际上，位图索引通常更强大些。表13-3总结两种类型的索引，以及它们处理不同类型条件的能力。下一节的目标是提供一些关于它们的例子。我也会介绍不同的性能和限制。

表13-3 可能导致Index Unique/Range Scan的条件

条 件	B树索引	位图索引
Equality (=)	✓	✓
IS NULL	✓*	✓
Range (BETWEEN、>、>=、<和<=)	✓	✓
IN	✓	✓
LIKE	✓	✓
Inequality (!=和<>) 和IS NOT NULL		✓†

* 不适用于单列索引；仅在另一个条件导致索引范围扫描或NULL值存储在索引中时适用于复合索引。

† 仅在多个位图索引组合时适用。

接下来的部分中许多例子都是基于conditions.sql脚本。Test表和它的索引使用以下SQL语句创建：

```
CREATE TABLE t (
  id NUMBER,
  d1 DATE,
  n1 NUMBER,
  n2 NUMBER,
  n3 NUMBER,
  n4 NUMBER,
  n5 NUMBER,
```

```
  n6 NUMBER,
  c1 VARCHAR2(20),
  c2 VARCHAR2(20),
  pad VARCHAR2(4000),
  CONSTRAINT t_pk PRIMARY KEY (id)
);

CREATE INDEX i_n1 ON t (n1);

CREATE INDEX i_n2 ON t (n2);

CREATE INDEX i_n3 ON t (n3);

CREATE INDEX i_n123 ON t (n1, n2, n3);

CREATE BITMAP INDEX i_n4 ON t (n4);

CREATE BITMAP INDEX i_n5 ON t (n5);

CREATE BITMAP INDEX i_n6 ON t (n6);

CREATE INDEX i_c1 ON t (c1);

CREATE BITMAP INDEX i_c2 ON t (c2);
```

3. 等式条件与B树索引

使用B树索引时，等式条件会使用其中一个操作执行。第一，INDEX UNIQUE SCAN，专门用来使用唯一索引。顾名思义，使用它最多返回一个rowid。下面的查询举例说明。执行计划证实通过操作2访问谓词，在id列上的条件使用了t_pk索引。接着，操作1使用从索引中提取的rowid来访问表。这是通过TABLE ACCESS BY INDEX ROWID操作来完成的。请注意这两个操作都是只执行一次：

```
SELECT /*+ index(t) */ * FROM t WHERE id = 6

------------------------------------------------------------------
| Id  | Operation                   | Name | Starts | A-Rows |
------------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |      1 |      1 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |      1 |      1 |
|*  2 |   INDEX UNIQUE SCAN         | T_PK |      1 |      1 |
------------------------------------------------------------------

  2 - access("ID"=6)
```

注意 本节内容，要强制索引扫描，只需指定表名来使用index hint。当这么使用时，查询优化器可以在所有可用的索引中任意选择。在所有例子中，单谓词存在于WHERE子句中。为此，查询优化器总是选择基于引用谓词列上的索引。

第二，INDEX RANGE SCAN，用来使用非唯一索引。这个操作与之前的唯一区别就是它可以提取许

多rowid（本例为527），而不仅是一个：

```
SELECT /*+ index_asc(t) */ * FROM t WHERE n1 = 6

----------------------------------------------------------------
| Id  | Operation                   | Name | Starts | A-Rows |
----------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |      1 |    527 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |      1 |    527 |
|*  2 |   INDEX RANGE SCAN          | I_N1 |      1 |    527 |
----------------------------------------------------------------

  2 - access("N1"=6)
```

从12.1版本开始，基于从索引范围扫描返回的rowid来访问表的操作是TABLE ACCESS BY INDEX ROWID BATCHED。这个操作的目的是利用访问多个rowid来优化表访问。下面的例子展示的执行计划与上一个查询一致：

```
SELECT /*+ index_asc(t) */ * FROM t WHERE n1 = 6

------------------------------------------------------------------------
| Id  | Operation                           | Name | Starts | A-Rows |
------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |      |      1 |    527 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| T    |      1 |    527 |
|*  2 |   INDEX RANGE SCAN                  | I_N1 |      1 |    527 |
------------------------------------------------------------------------

  2 - access("N1"=6)
```

默认情况下，索引扫描按照升序执行。因此，index hint命令优化器也按照这个顺序执行。要明确指定扫描顺序，可以使用index_asc和index_desc hint。下面的查询举例说明。在执行计划中，降序扫描显示为INDEX RANGE SCAN DESCENDING操作：

```
SELECT /*+ index_desc(t) */ * FROM t WHERE n1 = 6

-----------------------------------------------------------------
| Id  | Operation                    | Name | Starts | A-Rows |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |      1 |    527 |
|   1 |  TABLE ACCESS BY INDEX ROWID | T    |      1 |    527 |
|*  2 |   INDEX RANGE SCAN DESCENDING| I_N1 |      1 |    527 |
-----------------------------------------------------------------

  2 - access("N1"=6)
      filter("N1"=6)
```

请注意，INDEX RANGE SCAN和INDEX RANGE SCAN DESCENDING返回相同的数据。只是顺序不同。稍后，在范围条件部分会介绍何时会用到这样的访问路径。

4. 等式条件与位图索引

使用位图索引时，等式条件会使用三个操作。按照执行的次序，第一个操作是BITMAP INDEX SINGLE

VALUE，它会扫描索引并应用限制。顾名思义，这个操作寻找单个值。第二个操作是BITMAP CONVERSION TO ROWIDS，通过第一个操作获得的内容转换成rowid列表。第三个操作使用第二个操作建立的rowid列表来访问表。请注意这三个操作都只执行一次：

```
SELECT /*+ index(t i_n4) */ * FROM t WHERE n4 = 6

-----------------------------------------------------------------
| Id  | Operation                     | Name | Starts | A-Rows |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT              |      |      1 |    527 |
|   1 |  TABLE ACCESS BY INDEX ROWID  | T    |      1 |    527 |
|   2 |   BITMAP CONVERSION TO ROWIDS |      |      1 |    527 |
|*  3 |    BITMAP INDEX SINGLE VALUE  | I_N4 |      1 |      1 |
-----------------------------------------------------------------

  3 - access("N4"=6)
```

警告 在这一节里，要强制索引扫描，只需指定表名来使用index hint。换句话说，hint指定了该使用哪个索引。当这么使用时，hint只有在同名索引存在时才有效。由于索引名很容易改变（例如，使用ALTER INDEX RENAME语句），所以实际上hint也容易发生语法错误导致其失效。

对于B树索引来说，从12.1版本开始，使用索引返回的多个rowid访问表的操作是：TABLE ACCESS BY INDEX ROWID BATCHED。下面的例子展示的执行计划与上一个查询一致：

```
SELECT /*+ index(t i_n4) */ * FROM t WHERE n4 = 6

-------------------------------------------------------------------------
| Id  | Operation                            | Name | Starts | A-Rows |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT                     |      |      1 |    527 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED | T    |      1 |    527 |
|   2 |   BITMAP CONVERSION TO ROWIDS        |      |      1 |    527 |
|*  3 |    BITMAP INDEX SINGLE VALUE         | I_N4 |      1 |      1 |
-------------------------------------------------------------------------

  3 - access("N4"=6)
```

5. IS NULL条件与B树索引

使用B树索引，IS NULL条件仅能通过复合B树索引来使用，如果其他条件导致索引范围扫描或NULL值保存在索引中，那么这两个条件至少需要满足一个，因为仅有NULL值的索引项既不存在于单列索引中也不存在 于组合索引中。

下面的查询举例说明指定两个条件的案例。执行计划证实通过访问操作2的谓词，n2列上的条件使用i_n123索引。同时请注意操作2仅返回了5行，然而在“等式条件与B树索引”部分，并没有n2 IS NULL限制，范围扫描返回了527行：

```
SELECT /*+ index(t) */ * FROM t WHERE n1 = 6 AND n2 IS NULL
```

```
---------------------------------------------------------------
| Id  | Operation                   | Name   | Starts | A-Rows |
---------------------------------------------------------------
|   0 | SELECT STATEMENT            |        |      1 |      5 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T      |      1 |      5 |
|*  2 |   INDEX RANGE SCAN          | I_N123 |      1 |      5 |
---------------------------------------------------------------

  2 - access("N1"=6 AND "N2" IS NULL)
```

正如下面例子所示，同样的执行计划也用于当IS NULL条件指定在索引前导列上时：

```
SELECT /*+ index(t) */ * FROM t WHERE n1 IS NULL AND n2 = 8

---------------------------------------------------------------
| Id  | Operation                   | Name   | Starts | A-Rows |
---------------------------------------------------------------
|   0 | SELECT STATEMENT            |        |      1 |      4 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T      |      1 |      4 |
|*  2 |   INDEX RANGE SCAN          | I_N123 |      1 |      4 |
---------------------------------------------------------------

  2 - access("N1" IS NULL AND "N2"=8)
      filter("N2"=8)
```

其他IS NULL条件可以通过复合B树索引应用的情况，是当NULL值保存在索引中时。下面的查询就是这种情况。请注意，由于n2的IS NOT NULL条件，它保证了所有满足WHERE子句的行都会在i_n123索引中有对应的索引项。因此，可以使用索引范围扫描来找到它们：

```
SELECT /*+ index(t) */ * FROM t WHERE n1 IS NULL AND n2 IS NOT NULL

---------------------------------------------------------------
| Id  | Operation                   | Name   | Starts | A-Rows |
---------------------------------------------------------------
|   0 | SELECT STATEMENT            |        |      1 |    521 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T      |      1 |    521 |
|*  2 |   INDEX RANGE SCAN          | I_N123 |      1 |    521 |
---------------------------------------------------------------

  2 - access("N1" IS NULL)
      filter("N2" IS NOT NULL)
```

当至少有一列为非空时，NULL值也会存储在组合索引中。一个特别的案例满足这个条件，索引创建在可为空的列和常数上。不用说也知道这是圈套。但是在某些情况下，这是很有用的。下面的索引举例说明（请注意，可以使用另一个值来代替0）：

```
CREATE INDEX i_n1_nn ON t (n1, 0)
```

使用这样的索引，类似以下的查询能够执行索引范围扫描：

```
SELECT /*+ index(t) */ * FROM t WHERE n1 IS NULL

--------------------------------------------------------------------------
| Id  | Operation                   | Name    | Starts | E-Rows | A-Rows |
--------------------------------------------------------------------------
```

```
|   0 | SELECT STATEMENT            |         |      1 |        |    526 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T       |      1 |    526 |    526 |
|*  2 |   INDEX RANGE SCAN          | I_N1_NN |      1 |    526 |    526 |
--------------------------------------------------------------------------

  2 - access("N1" IS NULL)
```

然而，单列索引不能用于IS NULL条件。这是因为NULL值无法存储在索引中。因此，在本例中查询优化器无法利用索引范围扫描。即使尝试强制使用index hint，也会执行全表扫描或全索引扫描：

```
SELECT /*+ index(t i_n1) */ * FROM t WHERE n1 IS NULL

-----------------------------------------------------
| Id  | Operation         | Name | Starts | A-Rows |
-----------------------------------------------------
|   0 | SELECT STATEMENT  |      |      1 |    526 |
|*  1 |  TABLE ACCESS FULL| T    |      1 |    526 |
-----------------------------------------------------

  1 - filter("N1" IS NULL)
```

6. IS NULL条件与位图索引

使用位图索引，IS NULL条件使用与等式条件相同的方法执行。因为位图索引存储NULL值的方式与存储其他值的方式一致：

```
SELECT /*+ index(t i_n4) */ * FROM t WHERE n4 IS NULL

------------------------------------------------------------------
| Id  | Operation                    | Name | Starts | A-Rows |
------------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |      1 |    526 |
|   1 |  TABLE ACCESS BY INDEX ROWID | T    |      1 |    526 |
|   2 |   BITMAP CONVERSION TO ROWIDS|      |      1 |    526 |
|*  3 |    BITMAP INDEX SINGLE VALUE | I_N4 |      1 |      1 |
------------------------------------------------------------------

  3 - access("N4" IS NULL)
```

7. 范围条件与B树索引

使用B树索引，范围条件使用与等式条件在非唯一索引上相同的执行方法，或者换句话说，使用INDEX RANGE SCAN操作。对于范围条件来说，与其索引类型（即其独特性）无关。由于是范围扫描，总是可以返回多个rowid。例如，下面的查询显示范围条件应用在由主键构成的列上：

```
SELECT /*+ index(t (t.id)) */ * FROM t WHERE id BETWEEN 6 AND 19

-----------------------------------------------------------------
| Id  | Operation                   | Name | Starts | A-Rows |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |      1 |     14 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |      1 |     14 |
|*  2 |   INDEX RANGE SCAN          | T_PK |      1 |     14 |
-----------------------------------------------------------------
```

```
  2 - access("ID">=6 AND "ID"<=19)
```

注意　在这一节里，通过指定表名以及在哪一列创建索引，来使用多个hint强制索引扫描。对比这种语法与指定索引名的方法，优势在于hint并不需要依赖索引名。这让hint的可靠性更强。它的劣势是hint不能保证查询优化器总是选择相同的索引。

正如在上一部分提到的那样，索引扫描默认是升序执行的。这意味着将使用二进制比较（稍后会在“语言索引”部分提供关于不同类型的比较以及如何处理它们的信息）的ORDER BY作为范围条件应用于相同的列时，已对结果集进行排序。因此，就不需要再执行排序了。然而，当ORDER BY需要降序排列时，就需要明确执行排序，正如下面的查询所示。排序由操作1 SORT ORDER BY执行。请注意索引扫描被index_asc hint强制升序扫描：

```
SELECT /*+ index_asc(t (t.id)) */ * FROM t WHERE id BETWEEN 6 AND 19 ORDER BY id DESC

-------------------------------------------------------------------
| Id  | Operation                    | Name | Starts | A-Rows |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |      1 |     14 |
|   1 |  SORT ORDER BY               |      |      1 |     14 |
|   2 |   TABLE ACCESS BY INDEX ROWID| T    |      1 |     14 |
|*  3 |    INDEX RANGE SCAN          | T_PK |      1 |     14 |
-------------------------------------------------------------------

  3 - access("ID">=6 AND "ID"<=19)
```

同样的查询可以利用降序索引扫描来避免强制排序。下面的例子，执行计划中已经不存在SORT ORDER BY操作了：

```
SELECT /*+ index_desc(t (t.id)) */ * FROM t WHERE id BETWEEN 6 AND 19 ORDER BY id DESC

----------------------------------------------------------------------------
| Id  | Operation                    | Name | Starts | E-Rows | A-Rows |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |      1 |        |     14 |
|   1 |  TABLE ACCESS BY INDEX ROWID | T    |      1 |     15 |     14 |
|*  2 |   INDEX RANGE SCAN DESCENDING| T_PK |      1 |     15 |     14 |
----------------------------------------------------------------------------

  2 - access("ID"<=19 AND "ID">=6)
```

8. 范围条件与位图索引

使用位图索引，范围条件使用与等式条件类似的方法执行。唯一不同的是，BITMAP INDEX RANGE SCAN操作用来替代BITMAP INDEX SINGLE VALUE操作：

```
SELECT /*+ index(t (t.n4)) */ * FROM t WHERE n4 BETWEEN 6 AND 19

-------------------------------------------------------------------
| Id  | Operation                    | Name | Starts | A-Rows |
-------------------------------------------------------------------
```

```
|   0 | SELECT STATEMENT            |      |      1 |   6840 |
|   1 |  TABLE ACCESS BY INDEX ROWID | T    |      1 |   6840 |
|   2 |   BITMAP CONVERSION TO ROWIDS|      |      1 |   6840 |
|*  3 |    BITMAP INDEX RANGE SCAN   | I_N4 |      1 |     13 |
-----------------------------------------------------------------

  3 - access("N4">=6 AND "N4"<=19)
```

对于位图索引来说，没有升序和降序扫描的概念，因此无法避免和优化ORDER BY操作。

9. IN条件

IN条件并没有指定的访问路径。相反，在执行计划中，由于IN条件被执行了多次，INLIST ITERATOR操作指出执行计划的其中一部分。下面的三个查询展示了根据索引类型来使用索引扫描的操作。第一个是唯一索引，第二个是非唯一B树索引，第三个是位图索引。基本上，IN条件就是一系列的等式条件。请注意，操作相关的索引和表访问针对IN列表（请看Starts列）中的每个值只执行一次：

```
SELECT /*+ index(t t_pk) */ * FROM t WHERE id IN (6, 8, 19, 28)

-----------------------------------------------------------------
| Id  | Operation                   | Name | Starts | A-Rows |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |      1 |      4 |
|   1 |  INLIST ITERATOR            |      |      1 |      4 |
|   2 |   TABLE ACCESS BY INDEX ROWID| T    |      4 |      4 |
|*  3 |    INDEX UNIQUE SCAN         | T_PK |      4 |      4 |
-----------------------------------------------------------------

  3 - access(("ID"=6 OR "ID"=8 OR "ID"=19 OR "ID"=28))

SELECT /*+ index(t i_n1) */ * FROM t WHERE n1 IN (6, 8, 19, 28)

---------------------------------------------------------------------------
| Id  | Operation                   | Name | Starts | E-Rows | A-Rows |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |      1 |        |   1579 |
|   1 |  INLIST ITERATOR            |      |      1 |        |   1579 |
|   2 |   TABLE ACCESS BY INDEX ROWID| T    |      4 |   1710 |   1579 |
|*  3 |    INDEX RANGE SCAN          | I_N1 |      4 |   1710 |   1579 |
---------------------------------------------------------------------------

  3 - access(("N1"=6 OR "N1"=8 OR "N1"=19 OR "N1"=28))

SELECT /*+ index(t i_n4) */ * FROM t WHERE n4 IN (6, 8, 19, 28)

------------------------------------------------------------------
| Id  | Operation                    | Name | Starts | A-Rows |
------------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |      1 |   1579 |
|   1 |  INLIST ITERATOR             |      |      1 |   1579 |
|   2 |   TABLE ACCESS BY INDEX ROWID | T    |      4 |   1579 |
|   3 |    BITMAP CONVERSION TO ROWIDS|      |      4 |   1579 |
|*  4 |     BITMAP INDEX SINGLE VALUE | I_N4 |      4 |      3 |
------------------------------------------------------------------
```

```
4 - access(("N4"=6 OR "N4"=8 OR "N4"=19 OR "N4"=28))
```

多种表达式的动态条件

Oracle数据库不支持IN条件中有超过1000个表达式。即使可以使用多个分隔谓词来变通，但从性能的角度考虑，最好设置限制。因此，既不应该在IN条件中使用过多的表达式，也不应该使用分隔谓词来绕过1000个表达式的限制。实际上，应该尽可能避免长列表的表达式会带来的性能问题。相反，应该使用以下技巧之一。

- 基于读取（临时）表的子查询使用IN条件。
- 使用基于对象类型和为每个元素返回单行的嵌套表，作为输入的管道表函数的子查询使用IN条件。
- 使用MEMBER条件来测试一个元素是否是基于对象类型的嵌套表成员。

Dynamic_in_conditions.sql脚本为每一项技巧提供了例子。

10. LIKE条件

数据引擎能够使用LIKE条件作为访问谓词，但仅基于第一个通配符前的字符串。结果，提供的模式并不是从通配符开始的（下划线和百分号），LIKE条件与范围条件以相同的方式执行。此外，无法避免全表扫描或非全索引扫描。下面举例说明。前两个查询分别取回了c1列和c2列所有以字母A开头的行。因此，会执行范围扫描。第三个和第四个查询分别取回了c1列和c2列所有在任意位置包含字母A的行。因此会执行全索引扫描：

```
SELECT /*+ index(t i_c1) */ * FROM t WHERE c1 LIKE 'A%'

-----------------------------------------------------------------
| Id  | Operation                    | Name | Starts | A-Rows |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |      1 |    119 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |      1 |    119 |
|*  2 |   INDEX RANGE SCAN           | I_C1 |      1 |    119 |
-----------------------------------------------------------------

  2 - access("C1" LIKE 'A%')
      filter("C1" LIKE 'A%')

SELECT /*+ index(t i_c2) */ * FROM t WHERE c2 LIKE 'A%'

------------------------------------------------------------------
| Id  | Operation                     | Name | Starts | A-Rows |
------------------------------------------------------------------
|   0 | SELECT STATEMENT              |      |      1 |    108 |
|   1 |  TABLE ACCESS BY INDEX ROWID  | T    |      1 |    108 |
|   2 |   BITMAP CONVERSION TO ROWIDS|      |      1 |    108 |
|*  3 |    BITMAP INDEX RANGE SCAN    | I_C2 |      1 |    108 |
------------------------------------------------------------------
```

```
  3 - access("C2" LIKE 'A%')
      filter(("C2" LIKE 'A%' AND "C2" LIKE 'A%'))

SELECT /*+ index(t i_c1) */ * FROM t WHERE c1 LIKE '%A%'

-------------------------------------------------------------
| Id  | Operation                   | Name | Starts | A-Rows |
-------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |      1 |   1921 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |      1 |   1921 |
|*  2 |   INDEX FULL SCAN           | I_C1 |      1 |   1921 |
-------------------------------------------------------------

  2 - filter(("C1" LIKE '%A%' AND "C1" IS NOT NULL))

SELECT /*+ index(t i_c2) */ * FROM t WHERE c2 LIKE '%A%'

--------------------------------------------------------------
| Id  | Operation                    | Name | Starts | A-Rows |
--------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |      1 |   1846 |
|   1 |  TABLE ACCESS BY INDEX ROWID | T    |      1 |   1846 |
|   2 |   BITMAP CONVERSION TO ROWIDS|      |      1 |   1846 |
|*  3 |    BITMAP INDEX FULL SCAN    | I_C2 |      1 |   1846 |
--------------------------------------------------------------

  3 - filter(("C2" LIKE '%A%' AND "C2" IS NOT NULL))
```

11. 不等式与IS NOT NULL条件

正如表13-3所示，基于不等式（!=，<>）或IS NOT NULL条件无法使用索引范围扫描。为了举例说明这种限制并介绍如何优化这类SQL语句，让我们看一个基于inequalities.sql脚本的例子。Test表有一个分布不均匀的列叫作status。实际上，大部分行的status都设置为processed (P)。示例如下：

```
SQL> SELECT status, count(*)
  2  FROM t
  3  GROUP BY status;

S   COUNT(*)
- ----------
A          7
P     159981
R          4
X          8
```

一个应用选择所有status不是processed的列。为此，它执行以下查询：

```
SELECT * FROM t WHERE status != 'P'
```

即使查询有很强的选择性并且status列上有索引，查询优化器为了读取19行而选择全表扫描还是导致了实在太多的逻辑读（23063）：

```
-------------------------------------------------------------
| Id  | Operation          | Name | Starts | A-Rows | Buffers |
```

```
---------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |      19 |   23063 |
|*  1 |  TABLE ACCESS FULL| T    |     1 |      19 |   23063 |
---------------------------------------------------------------

   1 - filter("STATUS"<>'P')
```

在这样的例子中，不等式条件有很强的选择性，不过可以利用索引。可以使用以下三种技巧。

第一，如果不等式条件可以写进IN条件中，那么就可以使用索引范围扫描。这仅当已知选择值的数量并且数量有限时使用。下面的查询举例说明：

```
SELECT * FROM t WHERE status IN ('A','R','X')

-----------------------------------------------------------------------------
| Id  | Operation                    | Name     | Starts | A-Rows | Buffers |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |          |      1 |     19 |      13 |
|   1 |  INLIST ITERATOR             |          |      1 |     19 |      13 |
|   2 |   TABLE ACCESS BY INDEX ROWID| T        |      3 |     19 |      13 |
|*  3 |    INDEX RANGE SCAN          | I_STATUS |      3 |     19 |       7 |
-----------------------------------------------------------------------------

   3 - access(("STATUS"='A' OR "STATUS"='R' OR "STATUS"='X'))
```

第二，如果上一条技巧由于值未知或值的数量指定过高而不能使用，那么可以分成两个范围谓词重写不等式，结果就可以对它们每个执行一次索引范围扫描。可以考虑利用or扩展查询转换（请参考第6章关于它的信息）。查询重写后会像下面这样：

```
SELECT * FROM t WHERE status < 'P' OR status > 'P'

-----------------------------------------------------------------------------
| Id  | Operation                    | Name     | Starts | A-Rows | Buffers |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |          |      1 |     19 |      12 |
|   1 |  CONCATENATION               |          |      1 |     19 |      12 |
|   2 |   TABLE ACCESS BY INDEX ROWID| T        |      1 |      7 |       5 |
|*  3 |    INDEX RANGE SCAN          | I_STATUS |      1 |      7 |       3 |
|   4 |   TABLE ACCESS BY INDEX ROWID| T        |      1 |     12 |       7 |
|*  5 |    INDEX RANGE SCAN          | I_STATUS |      1 |     12 |       3 |
-----------------------------------------------------------------------------

   3 - access("STATUS"<'P')
   5 - access("STATUS">'P')
       filter(LNNVL("STATUS"<'P'))
```

万一or扩展并没有自动生效，可以手动重写查询来确保组件查询可以利用索引范围扫描并且将逻辑读减至最小：

```
SELECT * FROM t WHERE status < 'P'
UNION ALL
SELECT * FROM t WHERE status > 'P'

-----------------------------------------------------------------------------
| Id  | Operation                    | Name     | Starts | A-Rows | Buffers |
```

```
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |          |      1 |     19 |      12 |
|   1 |  UNION-ALL                   |          |      1 |     19 |      12 |
|   2 |   TABLE ACCESS BY INDEX ROWID| T        |      1 |      7 |       5 |
|*  3 |    INDEX RANGE SCAN          | I_STATUS |      1 |      7 |       3 |
|   4 |   TABLE ACCESS BY INDEX ROWID| T        |      1 |     12 |       7 |
|*  5 |    INDEX RANGE SCAN          | I_STATUS |      1 |     12 |       3 |
-----------------------------------------------------------------------------

   3 - access("STATUS"<'P')
   5 - access("STATUS">'P')
```

第三个技巧基于索引全扫描。要使用它，可以使用index hint强制执行索引全扫描。从性能角度考虑，正如下面的例子所示，这不是最优的办法。对于强选择性的查询，逻辑读数（299）和返回行数（19）的比例太高了：

```
SELECT /*+ index(t) */ * FROM t WHERE status != 'P'

----------------------------------------------------------------------------
| Id  | Operation                   | Name     | Starts | A-Rows | Buffers |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |          |      1 |     19 |     299 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T        |      1 |     19 |     299 |
|*  2 |   INDEX FULL SCAN           | I_STATUS |      1 |     19 |     293 |
----------------------------------------------------------------------------

   2 - filter("STATUS"<>'P')
```

要想以最优的方式执行索引全扫描，索引的大小应该尽可能小。要达到这个目的，有两个技巧可以使用。第一个的目的是定义函数索引（稍后，“函数索引”部分会提供这些索引的附加信息）来避免索引热门值。要使用这个技巧，索引和使用索引的SQL语句都需要修改。

- 创建函数索引来排除热门值（对于更复杂的条件，也可以使用CASE表达式或decode函数）：

```
CREATE INDEX i_status ON t (nullif(status, 'P'))
```

- 修改查询的谓词来使用索引：

```
SELECT * FROM t WHERE nullif(status, 'P') IS NOT NULL

----------------------------------------------------------------------------
| Id  | Operation                   | Name     | Starts | A-Rows | Buffers |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |          |      1 |     19 |       9 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T        |      1 |     19 |       9 |
|*  2 |   INDEX FULL SCAN           | I_STATUS |      1 |     19 |       3 |
----------------------------------------------------------------------------

   2 - filter("T"."SYS_NC00004$" IS NOT NULL)
```

第二个技巧的目的是使用NULL值替换最热门的值，从而避免大部分行被索引引用。但是请注意，这个技巧只对B树索引有效。要实现它，需要执行以下步骤。

- 使用NULL替换最热门的值：

```
UPDATE t SET status = NULL WHERE status = 'P'
```

- 重建索引以将其大小缩至最小：

```
ALTER INDEX i_status REBUILD
```

- 使用IS NOT NULL替换不等值：

```
SELECT * FROM t WHERE status IS NOT NULL

-------------------------------------------------------------------------------
| Id  | Operation                   | Name     | Starts | A-Rows | Buffers |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |          |      1 |     19 |      10 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T        |      1 |     19 |      10 |
|*  2 |   INDEX FULL SCAN           | I_STATUS |      1 |     19 |       3 |
-------------------------------------------------------------------------------

  2 - filter("STATUS" IS NOT NULL)
```

总之，正如上一例子所示，可以使用不等式或IS NOT NULL条件来高效执行SQL语句。但是，需要特别处理。

12. Min/Max函数

要高效执行包含min或max函数的查询，可以使用B树索引执行两个具体操作。第一个，INDEX FULL SCAN(MIN/MAX)，当查询没有指定范围条件时使用。然而，不要管它的名称，它执行的不是全索引扫描。它只是简单地获得索引键最左边或最右边的值：

```
SELECT /*+ index(t t_pk) */ min(id) FROM t

---------------------------------------------------------------
| Id  | Operation                  | Name | Starts | A-Rows |
---------------------------------------------------------------
|   0 | SELECT STATEMENT           |      |      1 |      1 |
|   1 |  SORT AGGREGATE            |      |      1 |      1 |
|   2 |   INDEX FULL SCAN (MIN/MAX)| T_PK |      1 |      1 |
---------------------------------------------------------------
```

第二个是INDEX RANGE SCAN(MIN/MAX)，当查询在使用函数的列上指定条件时使用：

```
SELECT /*+ index(t t_pk) */ min(id) FROM t WHERE id > 42

-----------------------------------------------------------------
| Id  | Operation                    | Name | Starts | A-Rows |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |      1 |      1 |
|   1 |  SORT AGGREGATE              |      |      1 |      1 |
|   2 |   FIRST ROW                  |      |      1 |      1 |
|*  3 |    INDEX RANGE SCAN (MIN/MAX)| T_PK |      1 |      1 |
-----------------------------------------------------------------

  3 - access("ID">42)
```

不幸的是，在同一个查询中同时使用这两个函数（min和max）时，无法使用这个优化技巧。在这种情况下，会执行索引全扫描。下面的查询举例说明：

```
SELECT /*+ index(t t_pk) */ min(id), max(id) FROM t

Plan hash value: 56794325

-----------------------------------------------------
| Id  | Operation          | Name | Starts | A-Rows |
-----------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |      1 |
|   1 |  SORT AGGREGATE    |      |      1 |      1 |
|   2 |   INDEX FULL SCAN  | T_PK |      1 |  10000 |
-----------------------------------------------------
```

对于位图索引来说，并没有特别的操作用来执行min和max函数。会使用与等式条件和范围条件相同的操作。

13. 函数索引

每次索引列作为参数传递给函数时，或涉及表达式时，SQL引擎就无法使用建立在对应列上的索引来做索引范围扫描。因此，其中要遵守的一个基本原则就是绝不修改WHERE子句中的索引列返回值。例如，如果在c1列上存在索引，像upper(c1) = 'SELDON'的限制就无法通过建立在c1列上的索引高效使用。这应该很明显，因为你仅可以搜索存储在索引中的值，而不是别的东西。下面的例子，与本节的其他例子一样，引用自fbi.sql脚本：

```
SQL> CREATE INDEX i_c1 ON t (c1);

SQL> SELECT * FROM t WHERE upper(c1) = 'SELDON';

-------------------------------------------------------
| Id  | Operation          | Name | E-Rows | A-Rows |
-------------------------------------------------------
|   0 | SELECT STATEMENT   |      |        |      4 |
|*  1 |  TABLE ACCESS FULL | T    |    100 |      4 |
-------------------------------------------------------

  1 - filter(UPPER("C1")='SELDON')
```

基本原则的一个例外是当约束确保索引包含了必要的信息时。举例说明这种情况，在c1列上有两个约束为查询优化器提供信息：

```
SQL> ALTER TABLE t MODIFY (c1 NOT NULL);

SQL> ALTER TABLE t ADD CONSTRAINT t_c1_upper CHECK (c1 = upper(c1));

SQL> SELECT * FROM t WHERE upper(c1) = 'SELDON';

--------------------------------------------------------------------
| Id  | Operation                    | Name | E-Rows | A-Rows |
--------------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |        |      4 |
|   1 |  TABLE ACCESS BY INDEX ROWID | T    |    100 |      4 |
|*  2 |   INDEX RANGE SCAN           | I_C1 |      4 |      4 |
--------------------------------------------------------------------
```

13

```
   2 - access("C1"='SELDON')
       filter(UPPER("C1")='SELDON')
```

显然，如果限制导致强选择性，你会想要利用索引。为了这个目的，如果不能修改WHERE子句或指定约束，可以创建函数索引。简单地说，这是创建在函数返回值或表达式结果上的索引。下面是例子：

```
SQL> CREATE INDEX i_c1_upper ON t (upper(c1));

SQL> SELECT * FROM t WHERE upper(c1) = 'SELDON';

-------------------------------------------------------------------
| Id  | Operation                   | Name       | E-Rows | A-Rows |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT            |            |        |      4 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T          |      4 |      4 |
|*  2 |   INDEX RANGE SCAN          | I_C1_UPPER |      4 |      4 |
-------------------------------------------------------------------

   2 - access(UPPER("C1")='SELDON')
```

警告　函数索引是使用文字作为参数的函数，将初始化参数cursor_sharing设置为force或similar时，查询优化器不会选择它。这是因为文字被绑定变量替代了。fbi_cs.sql脚本展示了这样的案例。

正如第8章中的“扩展的执行计划”部分介绍的那样，函数使用和WHERE子句中表达式的另一个相关问题是，查询优化器错估由行的源操作应用它们而产生的结果集基数。本节中的例子（请注意执行计划中的E-Rows和A-Rows）使用函数索引来举例说明，查询优化器也可以提高它作出的估量。并且这种提高可以与函数索引是否用来访问数据无关。可以有更准确的估量，因为每个函数索引都会将一个隐藏列添加到它创建的表上。由于列的统计信息和直方图也会为隐藏列收集，因此查询优化器获取到的附加信息需要函数索引才可以使用。重点也需要指出，在表级别上对于新的隐藏对象在函数索引创建时不会收集统计信息。只会自动收集索引统计信息。因此，在创建完新的函数索引时，不要忘记收集表级别的对象统计信息。

函数索引也可以创建PL/SQL的用户自定义函数。唯一的要求是函数必须定义为DETERMINISTIC。

警告　基于用户自定义函数的函数索引，当它们依赖的PL/SQL代码改变时并不会无效或标记为不可用。当然，可能会导致错误的结果。如果改变了这样的函数代码，应该立刻重建依赖的索引。fbi_udf.sql脚本中有这个操作的例子。

从11.1版本开始，为了避免索引或SQL语句中重复的函数或表达式，可以基于函数或表达式创建虚拟列。这样，因可以直接在虚拟列上创建，代码对定义来说就可以是透明的了。下面的例子展示如何添加、创建索引以及使用虚拟列将upper函数应用到c1列上：

```
SQL> ALTER TABLE t ADD (c1_upper AS (upper(c1)));

SQL> CREATE INDEX i_c1_upper ON t (c1_upper);
```

```
SQL> SELECT * FROM t WHERE c1_upper = 'SELDON';

-------------------------------------------------------------------
| Id  | Operation                   | Name       | E-Rows | A-Rows |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT            |            |        |      4 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T          |      4 |      4 |
|*  2 |   INDEX RANGE SCAN          | I_C1_UPPER |      4 |      4 |
-------------------------------------------------------------------

  2 - access("C1_UPPER"='SELDON')
```

本节中的例子虽然都是基于B树索引，但也都支持函数位图索引。

14. 语言索引

默认情况下，数据库引擎会执行二进制比较（binary comparision）来对比字符串。字符可以通过其二进制值来进行对比。因此，仅在每个对应的数字代码匹配时，才会认为两个字符串是相等的。

数据库引擎也能够执行语言对比（linguistic comparision）。使用这些对比，每个字符的数字代码并不需要完全相同。例如，可以设置数据库引擎识别小写和大写字符是相同的，或者没有重音的字符。要管理SQL操作的行为，可以使用初始化参数nls_comp。可以将其设置为以下值之一。

- binary：使用二进制对比。这是默认值。
- linguistic：使用语言对比。初始化参数nls_sort指定应用于对比的语言排列顺序（还有规则）。对应版本可以接受的值可以通过以下查询查到：

  ```
  SELECT value FROM v$nls_valid_values WHERE parameter = 'SORT'
  ```

- ansi：该值只对向下兼容有效。会使用linguistic来替代。

在实例和会话级别上可以设置动态初始化参数nls_comp和nls_sort。在会话级别上，可以通过ALTER SESSION语句设置它们；也可以在操作系统级别上，在客户端（例如，在Microsoft Windows注册表中）定义它们。请注意在客户端进行设置属于正常情况，不是例外。因此，客户端设置覆盖服务器端设置是很常见的。

举个例子，假设一张表保存以下数据（表和Test查询可以在linguistic_index.sql脚本中找到）：

```
SQL> SELECT c1 FROM t;
C1
----------
Leon
Léon
LEON
LÉON
```

默认情况下，执行二进制对比。要使用语言对比，需要将初始化参数nls_comp设置为linguistic，并且必须通过初始化参数nls_sort指定语言排列顺序（还有规则）。下面的例子使用generic_m，ISO标准的拉丁字符：

```
SQL> ALTER SESSION SET nls_comp = linguistic;

SQL> ALTER SESSION SET nls_sort = generic_m;
```

```
SQL> SELECT c1 FROM t WHERE c1 = 'LEON';

C1
----------
LEON
```

正如预料的那样，使用上面的设置没有什么特别的会发生。这个特性由两个扩展generic_m提供。第一个是generic_m_ci。正如下面的查询展示的那样，使用它进行对比不区分大小写：

```
SQL> ALTER SESSION SET nls_sort = generic_m_ci;

SQL> SELECT c1 FROM t WHERE c1 = 'LEON';

C1
----------
Leon
LEON
```

第二个是generic_m_ai。正如下面的查询展示的那样，使用它进行对比区分大小写和重音：

```
SQL> ALTER SESSION SET nls_sort = generic_m_ai;

SQL> SELECT c1 FROM t WHERE c1 = 'LEON';

C1
----------
Leon
Léon
LEON
LÉON
```

从功能的角度考虑，这个功能很好。通过设置两个初始化参数，可以控制SQL操作的行为。让我们来检查一下，将初始化参数nls_comp设置为linguistic时，执行计划是否会发生改变：

```
SQL> CREATE INDEX i_c1 ON t (c1);

SQL> ALTER SESSION SET nls_sort = generic_m_ai;

SQL> ALTER SESSION SET nls_comp = binary;

SQL> SELECT /*+ index(t) */ * FROM t WHERE c1 = 'LEON';

------------------------------------------
| Id  | Operation                   | Name |
------------------------------------------
|   0 | SELECT STATEMENT            |      |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |
|*  2 |   INDEX RANGE SCAN          | I_C1 |
------------------------------------------

   2 - access("C1"='LEON')

SQL> ALTER SESSION SET nls_comp = linguistic;

SQL> SELECT /*+ index(t) */ * FROM t WHERE c1 = 'LEON';
```

```
---------------------------------
| Id  | Operation          | Name |
---------------------------------
|   0 | SELECT STATEMENT   |      |
|*  1 |  TABLE ACCESS FULL| T    |
---------------------------------

  1 - filter(NLSSORT("C1",'nls_sort=''GENERIC_M_AI''')=HEXTORAW('022601FE02380232') )
```

显然，将初始化参数nls_comp设置为linguistic时，就不会再使用索引。输出的最后一行表明了原因。因为函数nlssort是隐式应用给索引列c1，所以不会在索引中进行查找。因此，出于这个目的，函数索引需要避免全表扫描。重点需要认识到，索引的定义必须包含与初始化参数nls_sort相同的值。因此，如果使用多种语言，就需要创建多个索引：

```
SQL> CREATE INDEX i_c1_linguistic ON t (nlssort(c1,'nls_sort=generic_m_ai'));

SQL> SELECT /*+ index(t) */ * FROM t WHERE c1 = 'LEON';

------------------------------------------------------
| Id  | Operation                   | Name            |
------------------------------------------------------
|   0 | SELECT STATEMENT            |                 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T               |
|*  2 |   INDEX RANGE SCAN          | I_C1_LINGUISTIC |
------------------------------------------------------

  2 - access(NLSSORT("C1",'nls_sort=''GENERIC_M_AI''')=HEXTORAW('022601FE02380232') )
```

在10.2版本中，为了应用LIKE操作还有另外一个限制，数据库引擎不能使用语言索引。换句话说，全索引扫描或全表扫描无法避免。该限制在11.1版本之后不再存在。

即使本节的例子都是基于B树索引，但也同样支持语言索引。

下面的例子展示了语言索引也可以用来避免ORDER BY操作：

```
SQL> SELECT /*+ index(t) */ * FROM t WHERE c1 BETWEEN 'L' AND 'M' ORDER BY c1;

------------------------------------------------------
| Id  | Operation                   | Name            |
------------------------------------------------------
|   0 | SELECT STATEMENT            |                 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T               |
|*  2 |   INDEX RANGE SCAN          | I_C1_LINGUISTIC |
------------------------------------------------------

  2 - access(NLSSORT("C1",'nls_sort=''GENERIC_M_AI''')>=HEXTORAW('0226') AND NLSSORT(
              "C1",'nls_sort=''GENERIC_M_AI''')<=HEXTORAW('0230') )
```

总之，语言对比是一个强大的特性，而它对SQL语句来说是透明的。然而，仅在一组适合的索引存在时，数据库引擎才可以高效使用它们。因为在客户端级别的设置会影响索引的使用，因此必须谨慎使用。

15. 复合索引

到目前为止，除了一个例外，我讨论了索引键只包含单独一列的索引。然而，索引键可以包含多列（对于B树索引来说限制是32，位图索引是30）。使用多列的索引叫作*复合索引*（composite index，有时也叫*组合索引*，concatenated index或*多列索引*，multicolumn index）。在这一点上，B树索引与位图索引完全不同。所以我会分开讨论它们。请注意，本节中的所有例子都基于composite_index.sql脚本。

- **B树索引**

复合索引的目的有两个。第一，它们可以用来实现由多列组成的主键或唯一索引约束。第二，它们能够应用由多个SQL条件使用AND组成的谓词。请注意，当多个SQL条件使用OR来组合时，无法高效使用复合索引！

自然地，重点是讨论如何使用复合索引来应用限制。下面查询的用途就是这个：

```
SELECT * FROM t WHERE n1 = 6 AND n2 = 42 AND n3 = 11
```

让我们来看看当使用单列索引时会发生什么。使用在n1列上创建的索引，索引扫描返回了527个rowid。因为索引值保存了n1列相关的数据，只有操作2的n1 = 6谓词可以通过索引访问。其他两个谓词被操作1应用为过滤器。由于操作2返回了很多rowid，执行计划一共生成了327个逻辑读。当取回单行时，这是无法接受的：

```
------------------------------------------------------------------------
| Id  | Operation                    | Name | Starts | A-Rows | Buffers |
------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |      1 |      1 |     327 |
|*  1 |  TABLE ACCESS BY INDEX ROWID| T    |      1 |      1 |     327 |
|*  2 |   INDEX RANGE SCAN           | I_N1 |      1 |    527 |       4 |
------------------------------------------------------------------------

   1 - filter(("N2"=42 AND "N3"=11))
   2 - access("N1"=6)
```

使用在n2列上创建的索引，情况基本与上个例子一致。唯一改进的是，索引扫描返回了更少的rowid（89）。因此一共执行的逻辑读数（85）也少得多：

```
------------------------------------------------------------------------
| Id  | Operation                    | Name | Starts | A-Rows | Buffers |
------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |      1 |      1 |      85 |
|*  1 |  TABLE ACCESS BY INDEX ROWID| T    |      1 |      1 |      85 |
|*  2 |   INDEX RANGE SCAN           | I_N2 |      1 |     89 |       4 |
------------------------------------------------------------------------

   1 - filter(("N3"=11 AND "N1"=6))
   2 - access("N2"=42)
```

使用在n3列上创建的索引，情况仍然与前两个很相似。实际上，索引扫描返回了许多rowid（164）。一共的逻辑读数（141）仍然太高：

```
------------------------------------------------------------------------
| Id  | Operation                    | Name | Starts | A-Rows | Buffers |
```

```
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |       |      1 |      1 |     141 |
|*  1 |  TABLE ACCESS BY INDEX ROWID| T     |      1 |      1 |     141 |
|*  2 |   INDEX RANGE SCAN          | I_N3  |      1 |    164 |       4 |
-------------------------------------------------------------------------------

  1 - filter(("N2"=42 AND "N1"=6))
  2 - access("N3"=11)
```

总之，这三个索引没有一个可以高效地应用谓词。这三个限制的选择性太高。观察到的与存储在数据字典中的对象统计信息一致。实际上，每一列非重复值的数量很低，请查看以下查询演示：

```
SQL> SELECT column_name, num_distinct
  2  FROM user_tab_columns
  3  WHERE table_name = 'T' AND column_name IN ('ID', 'N1', 'N2', 'N3');

COLUMN_NAME NUM_DISTINCT
----------- ------------
ID                 10000
N1                    18
N2                   112
N3                    60
```

在这样的情况下，在多个列上创建的单个索引应用各种条件会更高效。例如，下面的执行计划展示了如果在三列上创建一个复合索引时，会发生什么。重点需要知道使用这个索引，逻辑读数（4）会更低，因为索引扫描仅返回了满足整个WHERE子句的行（本例中仅返回一行）：

```
---------------------------------------------------------------------------------
| Id  | Operation                   | Name   | Starts | A-Rows | Buffers |
---------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |        |      1 |      1 |       4 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T      |      1 |      1 |       4 |
|*  2 |   INDEX RANGE SCAN          | I_N123 |      1 |      1 |       3 |
---------------------------------------------------------------------------------

  2 - access("N1"=6 AND "N2"=42 AND "N3"=11)
```

此时，重点需要认清，即使引用自WHERE子句中的列不都是索引创建的列，数据库引擎也能够执行索引范围扫描。基本要求是，应该将条件应用到索引键的引导列上。例如，使用上一例中的i_n123索引，列n2和n3的条件是可选的。下面的查询展示了在n2列上没有条件存在的例子：

```
SELECT * FROM t WHERE n1 = 6 AND n3 = 11

---------------------------------------------------------------------------------
| Id  | Operation                   | Name   | Starts | A-Rows | Buffers |
---------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |        |      1 |      8 |      12 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T      |      1 |      8 |      12 |
|*  2 |   INDEX RANGE SCAN          | I_N123 |      1 |      8 |       4 |
---------------------------------------------------------------------------------

  2 - access("N1"=6 AND "N3"=11)
      filter("N3"=11)
```

上一个执行计划需要知道，即使n3 = 11的谓词出现在访问谓词里，但所有满足n1 = 6的谓词索引键都需要访问到。一方面，这个方法是次优的，因为访问了索引不需要的部分。另一方面，正如上一个例子展示的那样，在索引扫描期间应用n3 = 11的谓词，作为过滤器要比在访问表时应用好得多。无论如何，为了获得最佳性能，谓词应该应用在索引的引导列上。

当没有条件在索引键的引导列上时，也有索引可以被（高效）使用的案例。这样的操作叫作*索引跳跃扫描*（index skip scan）。然而，只有在引导列的非重复值数量非常小时才可以使用，因为会对引导列的每个值单独进行索引范围扫描。下面的查询展示了这样的例子。请注意index_ss hint和INDEX SKIP SCAN操作：

```
SELECT /*+ index_ss(t i_n123) */ * FROM t WHERE n2 = 42 AND n3 = 11

-----------------------------------------------------------------------
| Id  | Operation                    | Name   | Starts | A-Rows | Buffers |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT             |        |      1 |      2 |      33 |
|   1 |  TABLE ACCESS BY INDEX ROWID | T      |      1 |      2 |      33 |
|*  2 |   INDEX SKIP SCAN            | I_N123 |      1 |      2 |      31 |
-----------------------------------------------------------------------

   2 - access("N2"=42 AND "N3"=11)
       filter(("N2"=42 AND "N3"=11))
```

因为“常规”索引扫描（使用INDEX SKIP SCAN DESCENDING操作）支持降序索引跳跃扫描，所以可以使用两个hint index_ss_asc和index_ss_desc来控制扫描的顺序。

说到复合索引，我觉得有必要提到在处理它们时我遇到的最常见错误，也是最常被问到的问题。错误与过度索引化（overindexation）有关。总是会错误地认为，仅当所有组成索引键的列在WHERE子句中时，数据库引擎才能使用索引。在已经看到的数个例子中，其实并不是这么回事。这个误解通常会导致在同一张表上创建使用相同引导列的数个索引，比如，一个索引使用n1、n2和n3列，而另一个索引使用n1和n3列。第二种通常是不需要的。请注意这个多余的索引会是个问题，因为它们不仅会降低SQL语句修改索引数据的速度，同时也浪费了不必要的空间。

最常被问到的问题是：如何选择列的顺序。比如，如果索引键由n1、n2和n3列组成，哪种顺序最好？当所有索引列都在WHERE子句中时，索引的效率与索引中列的顺序无关。因此，最好的顺序就是，当并非所有索引列都在WHERE子句中时，可以尽最大可能频繁地使用索引。换句话说，应该使索引可以供最大数量的SQL语句使用。要确保这种情况，列需要根据它们使用的频率来排序。尤其应该将引导列指定为WHERE子句中使用更频繁的那个（当然，理想状态）。每当多列使用相同频率时，有以下两个对立的方法可供选择。

- 引导列应该是预期提供最好选择性的列。如果只使用等式，就应该用非重复值数最高的那一列。如果使用范围条件，就与非重复值数无关了。比如，想象一下时间戳的案例：很可能非重复值数会很高。但是因为时间戳频繁用于范围谓词中，重要的是真实选择性而不是非重复值数。如果限制仅应用于特定的列，使用可以提供强选择性的引导列对以后的SQL语句会很有帮助。换句话说，最大可能地使查询优化器能够选择索引。
- 引导列应该是非重复值数最低的那列。这对提高索引压缩比很有帮助。

索引压缩

在B树索引和位图索引之间有一个很重要的不同，就是使用压缩技术存储在索引叶块中的键。位图索引总是使用压缩，而B树索引只在需要的时候才会使用压缩。

在一个未压缩的B树索引中，每个键都是完整存储的。换句话说，如果多个键有相同的值，会分别存储每个键值。因此，在未压缩的索引中，在同一个叶块上通常会存储很多相同的值。为了禁止这种重复的发生，可以在（重）建索引时使用COMPRESS参数来压缩索引键，并且可选择需要压缩的列数。例如，i_n123索引由三列组成：n1、n2和n3，使用COMPRESS 1，指定仅压缩n1列；使用COMPRESS 2，指定压缩n1和n2列；而使用COMPRESS 3，指定压缩全部三列。当不指定压缩的列数时，非唯一索引会压缩所有列，而唯一索引会压缩列数-1个列。

由于压缩是从左向右，列应该按照选择性从高到低排列来获得最好的压缩比例。然而，只有在无法阻止查询优化器使用索引时才会重排索引的列。

压缩B树索引默认不启用，这是因为它并不总能减小索引大小。实际上，在某些情况下，使用压缩或许会使索引增大！因此，应该仅在真正有益的时候启用压缩。你有两个选择来查看给定索引的预期压缩率。第一，可以创建索引不使用压缩，然后再压缩，接着对比大小。第二，可以使用ANALYZE INDEX语句执行分析来找出最佳压缩的列数和使用最佳压缩能节省的空间大小。下面的例子展示针对i_n123索引的分析。请注意分析的输出写入index_stats表中。本例中，你可以知道对两列使用压缩可以节省17%的空间：

```
SQL> ANALYZE INDEX i_n123 VALIDATE STRUCTURE;

SQL> SELECT opt_cmpr_count, opt_cmpr_pctsave FROM index_stats;

OPT_CMPR_COUNT OPT_CMPR_PCTSAVE
-------------- ----------------
             2               17
```

下面的SQL语句不仅展示了如何对i_n123索引实施压缩，也展示了如何查看压缩结果：

```
SQL> SELECT blocks FROM index_stats;

    BLOCKS
----------
        40

SQL> ALTER INDEX i_n123 REBUILD COMPRESS 2;

SQL> ANALYZE INDEX i_n123 VALIDATE STRUCTURE;

SQL> SELECT blocks FROM index_stats;

    BLOCKS
----------
        32
```

从性能的角度考虑，压缩索引的核心优势是因为它们更小的体积，不仅在执行索引范围扫描和索引全扫描时逻辑读更少了，而且它们也更可能缓存在缓冲区缓存中。然而，缺点是可能会增加遇到块争用的可能（这个主题会在第16章介绍）。

- **位图索引**

复合位图索引很少创建。因为多个索引可以高效地合并来应用限制。要想知道位图索引有多强大，让我们来看几个查询。

第一个查询利用AND合并三个等式条件来使用三个位图索引。请注意index_combine hint强制该类的执行计划。第一，操作4基于n5列在该列查找满足限制的行来扫描索引。作为结果的位图传给了操作3。接着操作5和6在n6列和n4列的索引上分别执行同样的扫描。一旦三个索引都完成扫描，操作3计算三组位图的AND操作。最后，操作2转换作为结果的位图为rowid列，接着操作1使用它们访问表：

```
SELECT /*+ index_combine(t i_n4 i_n5 i_n6) */ *
FROM t
WHERE n4 = 6 AND n5 = 42 AND n6 = 11

------------------------------------------------------------------------------
| Id  | Operation                     | Name | Starts | A-Rows | Buffers |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |      |      1 |      1 |       7 |
|   1 |  TABLE ACCESS BY INDEX ROWID  | T    |      1 |      1 |       7 |
|   2 |   BITMAP CONVERSION TO ROWIDS|      |      1 |      1 |       6 |
|   3 |    BITMAP AND                 |      |      1 |      1 |       6 |
|*  4 |     BITMAP INDEX SINGLE VALUE| I_N5 |      1 |      1 |       2 |
|*  5 |     BITMAP INDEX SINGLE VALUE| I_N6 |      1 |      1 |       2 |
|*  6 |     BITMAP INDEX SINGLE VALUE| I_N4 |      1 |      1 |       2 |
------------------------------------------------------------------------------

  4 - access("N5"=42)
  5 - access("N6"=11)
  6 - access("N4"=6)
```

对于第一个查询，有必要再向你展示一下使用复合位图索引的执行计划。正如你所见，逻辑读数并没有下降很多（4代替7）。即使它更好，但是这样的复合索引远不及三个单列索引灵活。这就是为什么实际中很少创建复合位图索引的原因：

```
--------------------------------------------------------------------------------
| Id  | Operation                     | Name   | Starts | A-Rows | Buffers |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |        |      1 |      1 |       4 |
|   1 |  TABLE ACCESS BY INDEX ROWID  | T      |      1 |      1 |       4 |
|   2 |   BITMAP CONVERSION TO ROWIDS|        |      1 |      1 |       3 |
|*  3 |    BITMAP INDEX SINGLE VALUE  | I_N456 |      1 |      1 |       3 |
--------------------------------------------------------------------------------

  3 - access("N4"=6 AND "N5"=42 AND "N6"=11)
```

第二个查询与第一个类似。唯一的不同是OR替代了AND。注意在执行计划中操作3仅有的改变：

```
SELECT /*+ index_combine(t i_n4 i_n5 i_n6) */ *
```

```
FROM t
WHERE n4 = 6 OR n5 = 42 OR n6 = 11

-------------------------------------------------------------------------------
| Id  | Operation                    | Name | Starts | A-Rows | Buffers |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |      1 |    767 |     420 |
|   1 |  TABLE ACCESS BY INDEX ROWID | T    |      1 |    767 |     420 |
|   2 |   BITMAP CONVERSION TO ROWIDS|      |      1 |    767 |       7 |
|   3 |    BITMAP OR                 |      |      1 |      1 |       7 |
|*  4 |     BITMAP INDEX SINGLE VALUE| I_N4 |      1 |      1 |       3 |
|*  5 |     BITMAP INDEX SINGLE VALUE| I_N6 |      1 |      1 |       2 |
|*  6 |     BITMAP INDEX SINGLE VALUE| I_N5 |      1 |      1 |       2 |
-------------------------------------------------------------------------------

  4 - access("N4"=6)
  5 - access("N6"=11)
  6 - access("N5"=42)
```

第三个查询与第一个类似。这次，唯一的不同是n4 != 6条件（替代n4 = 6）。由于执行计划有很大不同，让我们来详细看一下。最初，操作6基于n5列在该列查找满足n5 = 42条件的行来扫描索引。作为结果的位图传递给操作5。接着，操作7在n6列的索引上针对n6 = 11的条件执行同样的扫描。一旦两个索引扫描都完成，操作5计算两组位图的AND条件并传递作为结果的位图给操作4。接下来，操作8基于n4列在该列查找满足n4 = 6条件（这与在WHERE子句中指定的相反）的行来扫描索引。作为结果的位图传递给操作4，然后它会从操作5传递过来的位图中减掉它们。接着，操作9和3针对n4 IS NULL条件执行同样的扫描。这一步很必要，因为NULL值并不满足n4 != 6的条件。最后，操作2转换作为结果的位图为rowid列，结果操作1使用它们访问表：

```
SELECT /*+ index_combine(t i_n4 i_n5 i_n6) */ *
FROM t
WHERE n4 != 6 AND n5 = 42 AND n6 = 11

---------------------------------------------------------------------------------
| Id  | Operation                      | Name | Starts | A-Rows | Buffers |
---------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |      |      1 |      1 |       9 |
|   1 |  TABLE ACCESS BY INDEX ROWID   | T    |      1 |      1 |       9 |
|   2 |   BITMAP CONVERSION TO ROWIDS  |      |      1 |      1 |       8 |
|   3 |    BITMAP MINUS                |      |      1 |      1 |       8 |
|   4 |     BITMAP MINUS               |      |      1 |      1 |       6 |
|   5 |      BITMAP AND                |      |      1 |      1 |       4 |
|*  6 |       BITMAP INDEX SINGLE VALUE| I_N5 |      1 |      1 |       2 |
|*  7 |       BITMAP INDEX SINGLE VALUE| I_N6 |      1 |      1 |       2 |
|*  8 |      BITMAP INDEX SINGLE VALUE | I_N4 |      1 |      1 |       2 |
|*  9 |     BITMAP INDEX SINGLE VALUE  | I_N4 |      1 |      1 |       2 |
---------------------------------------------------------------------------------

  6 - access("N5"=42)
  7 - access("N6"=11)
  8 - access("N4"=6)
  9 - access("N4" IS NULL)
```

总之，位图索引可以高效地合并，而且在合并期间还可以使用多个SQL条件。总而言之，它们非常灵活。由于这些特性，它们对报告系统非常重要，因为那里的查询都无法预先知道（固定）。

16. B树索引的位图计划

上节介绍的位图计划执行得很好，它们也可以应用在B树索引上。数据库引擎能够基于B树索引扫描返回的数据，创建一种内存中的位图索引。下面的查询，与在复合B树索引部分使用的一样。请注意执行计划中的BITMAP CONVERSION FROM ROWIDS操作负责转换：

```
SELECT /*+ index_combine(t i_n1 i_n2 i_n3) */ *
FROM t
WHERE n1 = 6 AND n2 = 42 AND n3 = 11

-------------------------------------------------------------------------------
| Id  | Operation                        | Name | Starts | A-Rows | Buffers |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                 |      |      1 |      1 |      10 |
|   1 |  TABLE ACCESS BY INDEX ROWID     | T    |      1 |      1 |      10 |
|   2 |   BITMAP CONVERSION TO ROWIDS    |      |      1 |      1 |       9 |
|   3 |    BITMAP AND                    |      |      1 |      1 |       9 |
|   4 |     BITMAP CONVERSION FROM ROWIDS|      |      1 |      1 |       3 |
|*  5 |      INDEX RANGE SCAN            | I_N2 |      1 |     89 |       3 |
|   6 |     BITMAP CONVERSION FROM ROWIDS|      |      1 |      1 |       3 |
|*  7 |      INDEX RANGE SCAN            | I_N3 |      1 |    164 |       3 |
|   8 |     BITMAP CONVERSION FROM ROWIDS|      |      1 |      1 |       3 |
|*  9 |      INDEX RANGE SCAN            | I_N1 |      1 |    527 |       3 |
-------------------------------------------------------------------------------

  5 - access("N2"=42)
  7 - access("N3"=11)
  9 - access("N1"=6)
```

注意　B树索引的位图计划也与位图索引一样，只能在企业版中使用。

17. 仅索引扫描

一个与索引相关的优化技巧是数据库引擎不仅可以从索引中提取rowid列来访问表，还可以提取存储在索引中的列数据。因此，当索引包含了所有查询需要处理的数据时，就会执行仅索引扫描（index-only scan）。这对减少逻辑读数很有帮助。实际上，仅索引扫描不访问表。当索引的群集因子高时，这对索引范围扫描非常有帮助。下面的查询举例说明。请注意没有执行访问表的操作：

```
SELECT c1 FROM t WHERE c1 LIKE 'A%'

-----------------------------------------------------------------
| Id  | Operation         | Name | Starts | A-Rows | Buffers |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |      1 |    119 |      11 |
|*  1 |  INDEX RANGE SCAN| I_C1 |      1 |    119 |      11 |
-----------------------------------------------------------------
```

```
  1 - access("C1" LIKE 'A%')
      filter("C1" LIKE 'A%')
```

如果SELECT子句引用n1列而不是c1列，那么查询优化器就无法利用仅索引扫描。请注意，在下面的例子中，查询是如何执行了130个逻辑读（针对索引执行了11个，针对表执行了119个，换句话说，每个从索引中获得rowid执行一个逻辑读）以取回119行的：

```
SELECT n1 FROM t WHERE c1 LIKE 'A%'

-----------------------------------------------------------------------
| Id  | Operation                   | Name | Starts | A-Rows | Buffers |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |      1 |    119 |     130 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |      1 |    119 |     130 |
|*  2 |   INDEX RANGE SCAN          | I_C1 |      1 |    119 |      11 |
-----------------------------------------------------------------------

  2 - access("C1" LIKE 'A%')
      filter("C1" LIKE 'A%')
```

在这类情况下，为了使用仅索引扫描，可以给索引增加列，使它们不会用来应用限制。理想做法是使用一个索引键创建组合索引来包含所有SQL语句引用到的列（也称为*覆盖索引*），而不仅仅是在WHERE子句中使用的列。换句话说，你“滥用”索引来存储多余的数据，从而减小逻辑读数。注意，不管怎样索引，引导列必须是WHERE子句引用的其中一列。在本例中，这代表在c1和n1列上创建复合索引。使用这个索引，同样的查询取回同样多的行只需要10个逻辑读而不是130个：

```
SELECT n1 FROM t WHERE c1 LIKE 'A%'

-------------------------------------------------------------
| Id  | Operation        | Name    | Starts | A-Rows | Buffers |
-------------------------------------------------------------
|   0 | SELECT STATEMENT |         |      1 |    119 |      10 |
|*  1 |  INDEX RANGE SCAN| I_C1N1  |      1 |    119 |      10 |
-------------------------------------------------------------

  1 - access("C1" LIKE 'A%')
      filter("C1" LIKE 'A%')
```

警告 对于列表分区表来说，仅在分区键是索引的一部分时，查询优化器基于仅索引扫描生成的执行计划来分解IN条件。index_only_scan_list_part.sql脚本提供了这样的例子。对于范围和散列分区表来说，这个限制不存在。

即使本节的例子都是基于B树索引，仅索引扫描也可以用于位图索引。

18. 索引组织表

创建索引组织表（index-organized table）是一种实现仅索引扫描的特殊方式。实际上，这类表的核心概念是为了彻底避免产生表段。相反，所有数据都会存储在基于主键的索引段上。同样也可以将部分数据存储在溢出段（overflow segment）上。然而一般来说，使用索引组织表的好处已经不存在（除

非溢出段很少被访问）。当创建辅助索引（secondary index，除了主键之外的另一个索引）时也会发生相同的事：需要访问两个段。因此没有必要使用它。由于这些原因，仅当满足两个需求时才会考虑使用索引组织表。第一，表通常使用主键访问。第二，所有数据可以存储在索引结构中（一行最多可以使用块的50%）。除此之外，没有必要使用它。

*物理*rowid（physical rowid）并不会引用索引组织表中的行。相反，它由*逻辑*rowid（logical rowid）引用。这类rowid由两部分组成：第一，对包含插入时行（键）的块推测。第二，主键的值。随着第一次推测，逻辑rowid会访问索引组织表，希望能找到行插入时所在的块，但是由于推测并不会在发生块分裂时更新，它有可能会在INSERT和UPDATE语句执行时变旧。如果推测正确，使用逻辑rowid，访问一行数据只需一个逻辑读。万一推测是错误的，逻辑读数会等于或者大于2（一个是通过推测无用的访问，加上使用主键的普通访问）。自然地，为了最好的性能，重要的是正确的推测。要评估推测的正确性，可以使用user_indexes（dba、all和在12.1多租户环境下的cdb版本的视图也包含pct_direct_access列）视图中的pct_direct_access列，它会由dbms_stats包来更新。这个值提供了针对某一索引推测正确的百分比。下面的例子，引用自iot_guess.sql脚本，不仅展示了过久的推测会影响逻辑读数，也展示了如何改变这种次优的情况（请注意，例子中使用的索引是辅助索引）：

```
SQL> SELECT pct_direct_access
  2  FROM user_indexes
  3  WHERE table_name = 'T' AND index_name = 'I';

PCT_DIRECT_ACCESS
-----------------
               76

SQL> SELECT count(pad) FROM t WHERE n > 0;

----------------------------------------------------------------
| Id  | Operation          | Name | Starts | A-Rows | Buffers |
----------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |      1 |    1496 |
|   1 |  SORT AGGREGATE    |      |      1 |      1 |    1496 |
|*  2 |   INDEX UNIQUE SCAN| T_PK |      1 |   1000 |    1496 |
|*  3 |    INDEX RANGE SCAN| I    |      1 |   1000 |       6 |
----------------------------------------------------------------

  2 - access("N">0)
  3 - access("N">0)

SQL> ALTER INDEX i UPDATE BLOCK REFERENCES;

SQL> execute dbms_stats.gather_index_stats(ownname => user, indname => 'i')

SQL> SELECT pct_direct_access
  2  FROM user_indexes
  3  WHERE table_name = 'T' AND index_name = 'I';

PCT_DIRECT_ACCESS
-----------------
              100

SQL> SELECT count(pad) FROM t WHERE n > 0;
```

```
-----------------------------------------------------------------
| Id  | Operation           | Name | Starts | A-Rows | Buffers |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |      1 |      1 |    1006 |
|   1 |  SORT AGGREGATE     |      |      1 |      1 |    1006 |
|*  2 |   INDEX UNIQUE SCAN| T_PK |      1 |   1000 |    1006 |
|*  3 |    INDEX RANGE SCAN| I    |      1 |   1000 |       6 |
-----------------------------------------------------------------

  2 - access("N">0)
  3 - access("N">0)
```

逻辑rowid一个有趣的副作用是辅助索引总会包含主键，即使它没有明确编入索引。下面的例子举例说明只靠仅索引扫描，数据库引擎如何从在另一列（n）上创建的辅助索引中提取主键（id）：

```
SQL> SELECT id FROM t WHERE n = 42;

----------------------------------
| Id  | Operation          | Name |
----------------------------------
|   0 | SELECT STATEMENT   |      |
|*  1 |  INDEX RANGE SCAN| I    |
----------------------------------

  1 - access("N"=42)
```

除了避免访问表段外，另外索引组织表提供了两个不应该低估的优势。第一，数据总是聚合的，因此基于主键的范围扫描总是可以高效执行，而不必像堆表那样只有在群集因子低的时候才可以。第二个优势是基于主键的范围扫描总是按照存储在主键索引中的数据顺序返回数据。这可以用来优化 ORDER BY 操作。

19. 全局、本地或非分区索引

使用分区表，通常会创建本地分区索引。这么做的主要优势是减少索引与表分区之间的依赖性。例如，当分区增加、删除、截断或交换时，它会使事情变得简单。简单来说，创建本地索引通常来说是有好处的。然而，也存在不能或者不建议这么做的情况。

前缀与非前缀索引

如果分区键是索引列的左前缀，那么这个索引就是前缀的，并且对于子分区索引，子分区键包含在索引键中。而本地索引可以是前缀或者非前缀的，只能创建全局前缀索引。

根据*Oracle Database VLDB and Partitioning Guide*手册，非前缀索引与前缀索引表现不同。实际上，我从未见过因为索引是非前缀的而导致的性能问题。我的建议是，创建最明智的索引而不用考虑它是否是前缀索引。

第一个问题与主键和唯一索引有关。实际上，基于本地索引，它们的键必须包含主键。尽管有些时候可行，通常有改变数据库逻辑设计的可能。这尤其是在使用范围分区时。因此，在我看来，这应该作为最后的手段。永远不应该搞乱逻辑设计。因为逻辑设计不能更改，仅剩下其他两种可能。第一

是创建非分区索引。第二是创建全局分区索引。后者仅在真正有优势时才会使用。因为这样的索引通常是散列分区，然而，仅在索引非常大或者索引经历非常高的负载时才值得这么做。总之，为了支持主键和唯一索引而创建非分区索引并不常见。

本地分区索引的第二个问题是对于不能利用分区裁剪的SQL语句，它们可以使性能变得糟糕。在本章之前的“范围分区”部分中描述过导致这种情况的原因。它对索引扫描的影响或许会非常高。下面的例子，基于图13-5的范围分区表，展示了可能会有的问题。首先，创建非分区索引。使用它查询，返回一行执行了4个逻辑读。请注意，TABLE ACCESS BY GLOBAL INDEX ROWID操作表明rowid来自全局或非分区索引：

```
SQL> CREATE INDEX i ON t (n3);

SQL> SELECT * FROM t WHERE n3 = 3885;

-----------------------------------------------------------------------------
| Id | Operation                           | Name | Starts | A-Rows | Buffers |
-----------------------------------------------------------------------------
|  0 | SELECT STATEMENT                    |      |      1 |      1 |       4 |
|  1 |  TABLE ACCESS BY GLOBAL INDEX ROWID| T    |      1 |      1 |       4 |
|* 2 |   INDEX RANGE SCAN                 | I    |      1 |      1 |       3 |
-----------------------------------------------------------------------------

   2 - access("N3"=3885)
```

对于该实验的第二部分，重建了索引。这次是本地索引。由于表有48个分区，所以索引也有48个分区。因为实验查询并不包含基于分区键上的限制，所以不会发生分区裁剪。这可以由PARTITION RANGE ALL操作以及Pstart和Pstop列来证实。同样注意到TABLE ACCESS BY LOCAL INDEX ROWID操作表明rowid来自本地分区索引。执行计划的问题是代替像上个案例那样执行单索引扫描，这次索引扫描是针对每个分区执行的（注意操作2和操作3的Starts列）。因此，即使只取回了一行，50个逻辑读也是必要的：

```
SQL> CREATE INDEX i ON t (n3) LOCAL;

SQL> SELECT * FROM t WHERE n3 = 3885;

-----------------------------------------------------------------------------------------------
| Id | Operation                           | Name | Starts | Pstart| Pstop | A-Rows | Buffers |
-----------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT                    |      |      1 |       |       |      1 |      50 |
|  1 |  PARTITION RANGE ALL                |      |      1 |     1 |    48 |      1 |      50 |
|  2 |   TABLE ACCESS BY LOCAL INDEX ROWID| T    |     48 |     1 |    48 |      1 |      50 |
|* 3 |    INDEX RANGE SCAN                | I    |     48 |     1 |    48 |      1 |      49 |
-----------------------------------------------------------------------------------------------

   3 - access("N3"=3885)
```

总之，不使用分区裁剪，逻辑读数会按照分区数成比例增长。因此，正如之前指出的，有时使用非分区索引要比分区索引好。或者，作为折中方案，应该限制分区数量。注意，有时你没有选择。比如，位图索引仅可以作为本地索引创建。

20. 不可见索引

从11.1版本之后，有个索引属性可以用来指定索引是否对查询优化器可见。默认情况下，索引是可见的。万一索引不可见，当索引基于的数据表发生修改时，就需要常规维护了，但是查询优化器无法在执行计划生成期间利用它。因为不可见索引是定期维护的，基于唯一索引的约束仍然会定期执行，即使它们基于的索引不可见。

警告　在11.1版本中，索引的不可见并不是全部。实际上，在两个突发情况下查询优化器可以利用不可见索引。第一，即使不可见索引不会包含在执行计划中，查询优化器也可以使用统计信息关联它来提高其估值。invisible_index_stats.sql脚本示范了这样的案例。第二，对于未加索引的外键，数据库引擎能够利用不可见索引来避免错误的争用。

下面的例子基于invisible_index.sql脚本，展示了如何使索引不可见以及这样的操作如何对指定查询产生影响：

```
SQL> SELECT * FROM t WHERE id = 42;

--------------------------------------------
| Id  | Operation                   | Name |
--------------------------------------------
|   0 | SELECT STATEMENT            |      |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |
|*  2 |   INDEX UNIQUE SCAN         | T_PK |
--------------------------------------------

  2 - access("ID"=42)

SQL> SELECT visibility FROM user_indexes WHERE index_name = 'T_PK';

VISIBILITY
----------
VISIBLE

SQL> ALTER INDEX t_pk INVISIBLE;

SQL> SELECT visibility FROM user_indexes WHERE index_name = 'T_PK';

VISIBILITY
----------
INVISIBLE

SQL> SELECT * FROM t WHERE id = 42;

----------------------------------
| Id  | Operation         | Name |
----------------------------------
|   0 | SELECT STATEMENT  |      |
|*  1 |  TABLE ACCESS FULL| T    |
----------------------------------

  1 - filter("ID"=42)
```

在以下两种情况下调整或创建不可见索引是有益的。

- 无论是否删除已存在的索引都不会影响访问性能。这在需要删除的索引很大时很有用。实际上，无法实验删除一个大索引，当你事后发现删除索引是个错误的时候，将需要花费太长时间和太多资源来重建它。
- 创建索引但不使它会立即对查询优化器生效。

默认情况下，查询优化器承认索引的可见性。这是因为，默认情况下会将初始化参数optimizer_use_invisible_indexes设置为FALSE。如果在系统级别或者会话级别上将这个参数设置为TRUE，查询优化器就允许不可见索引为可见。自从11.1.0.7版本开始，也可以在SQL语句中增加（no_）use_invisible_indexes hint来控制查询优化器是否承认索引的可见性。

警告　*根据High Availabilty Overview手册："任何DML操作可维护不可见索引，但是除非你明确使用hint指定索引，否则优化器并不会使用它。"不幸的是，这句话有错误。问题是index hint无法用来改变不可见索引的可见度。仅(no_)use_invisible_indexes hint可以影响不可见索引的可见度。*

直到11.2版本（包括该版本），在同一组列上不能创建多个索引（如果你尝试这么做，数据库引擎会报ORA-01408）。从12.1版本开始，这个限制被取消了。实际上，在同一组列上创建多个索引，在同一时间只有其中一个索引可见。这种可能性对于应用来说必须处理成高可用，这在改变索引的唯一性、类型（B树或位图）和分区而不需要计划停机时间时很有用。不使用这个特性，当删除和重建索引时或许会需要停止其他的高可用应用。下面的例子，引用自multiple_indxes.sql脚本的输出，举例说明这个特性。

(1) 设置初始化对象：

```
SQL> CREATE TABLE t (n1 NUMBER, n2 NUMBER, n3 NUMBER);

SQL> CREATE INDEX i_i ON t (n1);
```

(2) 在同样的列（n1）上不支持创建与上一个一致的不可见索引（请注意新索引是唯一的）：

```
SQL> CREATE UNIQUE INDEX i_ui ON t (n1);
CREATE UNIQUE INDEX i_ui ON t (n1)
                               *
ERROR at line 1:
ORA-01408: such column list already indexed
```

(3) 可以创建多个不可见索引（请注意每个索引的不同）：

```
SQL> CREATE UNIQUE INDEX i_ui ON t (n1) INVISIBLE;

SQL> CREATE BITMAP INDEX i_bi ON t (n1) INVISIBLE;

SQL> CREATE INDEX i_hpi ON t (n1) INVISIBLE
  2  GLOBAL PARTITION BY HASH (n1) PARTITIONS 4;

SQL> CREATE INDEX i_rpi ON t (n1) INVISIBLE
  2  GLOBAL PARTITION BY RANGE (n1) (
  3    PARTITION VALUES LESS THAN (10),
  4    PARTITION VALUES LESS THAN (MAXVALUE)
  5  );
```

(4) 通过使旧索引不可见并使新索引可见，在两个索引之间切换：

```
SQL> ALTER INDEX i_i INVISIBLE;

SQL> ALTER INDEX i_ui VISIBLE;
```

21. 局部索引

出于性能考虑，有时并不需要将表中的所有数据索引化。尤其是大范围分区表包含很长的特定历史时间数据（比如订单或通话记录）时。例如，可能仅需要对前一天的数据创建索引，或最近一周的数据，并且把所有过期的数据从索引中删除。这样的索引叫作*局部索引*（partial index）。在正确的场合使用它们可以节省许多不必要分配的磁盘空间。

即使一直到11.2版本（包括该版本），使用一些特别的技巧也可以使用一些局部索引，仅从12.1版本以后，Oracle数据库提供正规语法来支持局部索引。12.1版本引入语法的基本概念，可以在表或者分区级别设置数据是否使用索引。

下面的例子基于partial_index.sql脚本，展示了如何禁用除了设置INDEXING ON属性以外的其他所有分区的索引功能（显然你也可以在表级别设置INDEXING ON，并为指定分区设置INDEXING OFF）：

```
CREATE TABLE t (
  id NUMBER NOT NULL,
  d DATE NOT NULL,
  n NUMBER NOT NULL,
  pad VARCHAR2(4000) NOT NULL
)
INDEXING OFF
PARTITION BY RANGE (d) (
  PARTITION t_jan_2014 VALUES LESS THAN (to_date('2014-02-01','yyyy-mm-dd')),
  PARTITION t_feb_2014 VALUES LESS THAN (to_date('2014-03-01','yyyy-mm-dd')),
  PARTITION t_mar_2014 VALUES LESS THAN (to_date('2014-04-01','yyyy-mm-dd')),
  PARTITION t_apr_2014 VALUES LESS THAN (to_date('2014-05-01','yyyy-mm-dd')),
  PARTITION t_may_2014 VALUES LESS THAN (to_date('2014-06-01','yyyy-mm-dd')),
  PARTITION t_jun_2014 VALUES LESS THAN (to_date('2014-07-01','yyyy-mm-dd')),
  PARTITION t_jul_2014 VALUES LESS THAN (to_date('2014-08-01','yyyy-mm-dd')),
  PARTITION t_aug_2014 VALUES LESS THAN (to_date('2014-09-01','yyyy-mm-dd')),
  PARTITION t_sep_2014 VALUES LESS THAN (to_date('2014-10-01','yyyy-mm-dd')),
  PARTITION t_oct_2014 VALUES LESS THAN (to_date('2014-11-01','yyyy-mm-dd')),
  PARTITION t_nov_2014 VALUES LESS THAN (to_date('2014-12-01','yyyy-mm-dd')),
  PARTITION t_dec_2014 VALUES LESS THAN (to_date('2015-01-01','yyyy-mm-dd')) INDEXING ON
)
```

当索引创建后，可以指定是遵守索引属性（INDEXING PARTIAL）还是不遵守（INDEXING FULL，这是默认值）。下面的SQL语句展示如何创建局部索引：

```
CREATE INDEX i ON t (d) INDEXING PARTIAL
```

使用局部索引的关键要求是需要将数据存储在分区表中。索引是未分区、本地还是全局的都没关系。除了这些之外，只有使用INDEXING ON存储在分区中的行才会被索引。使用像上个例子那样创建的表和索引，查询优化器不会限制通过全表扫描或索引扫描来访问所有数据。相反，它可以利用表扩展查询转换（请参考第6章），从而基于数据是否有索引来生成不同的访问路径。下面的例子图示了这一情形。

```
SQL> SELECT *
  2  FROM t
  3  WHERE d BETWEEN to_date('2014-11-30 23:00:00','yyyy-mm-dd hh24:mi:ss')
  4              AND to_date('2014-12-01 01:00:00','yyyy-mm-dd hh24:mi:ss');

-------------------------------------------------------------------------------
| Id  | Operation                                  | Name    | Pstart| Pstop |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                           |         |       |       |
|   1 |  VIEW                                      | VW_TE_2 |       |       |
|   2 |   UNION-ALL                                |         |       |       |
|   3 |    TABLE ACCESS BY GLOBAL INDEX ROWID BATCHED| T       |    12 |    12 |
|*  4 |     INDEX RANGE SCAN                       | I       |       |       |
|   5 |    PARTITION RANGE SINGLE                  |         |    11 |    11 |
|*  6 |     TABLE ACCESS FULL                      | T       |    11 |    11 |
-------------------------------------------------------------------------------

   4 - access("T"."D">=TO_DATE(' 2014-12-01 00:00:00', 'syyyy-mm-dd
              hh24:mi:ss') AND "D"<=TO_DATE(' 2014-12-01 01:00:00', 'syyyy-mm-dd
              hh24:mi:ss'))
   6 - filter("D">=TO_DATE(' 2014-11-30 23:00:00', 'syyyy-mm-dd
              hh24:mi:ss'))
```

13.3.3　单表散列群集访问

实践中，很少有数据库使用单表散列群集。事实上，当它们按照正确的大小排列并通过群集键上的等式条件访问时，它们能提供非常好的性能。这有两个原因。第一，它们不需要分离的访问结构（比如，索引）来定位数据。实际上，群集键就足够用来定位它。第二，所有关联群集键的数据都聚合在一起。这两个优势也在本章之前部分的图13-3和图13-4中通过实验演示过。

单表散列群集用来实现通过指定键频繁地（理想上，总是）查找表。基本上可以在索引组织表上使用同样的方法。然而，它们之间有些主要区别。表13-4列出了单表散列群集与索引组织表相比的主要优势和劣势。主要的劣势是单表散列群集需要准确设置大小才能使用。

表13-4　单表散列群集与索引组织表的比较

优　　势	劣　　势
更好的性能（如果通过群集键访问并正确设置大小）	需要设置大小，忽略散列冲突并且浪费空间
群集键或许与主键不同	不支持分区
	不支持LOB列

当单表散列群集通过群集键访问时，执行计划中会出现TABLE ACCESS HASH操作。它通过群集键直接访问包含需要数据的块（也可能是多个）。下面引用自hash_cluster.sql脚本的数据，举例说明：

```
SELECT * FROM t WHERE id = 6

------------------------------------------------------------
| Id  | Operation         | Name | Starts | A-Rows | Buffers |
------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |      1 |      1 |       1 |
```

```
|* 1 |  TABLE ACCESS HASH| T    |      1 |      1 |      1 |
-----------------------------------------------------------

  1 - access("ID"=6)
```

除了等式条件，其他条件允许通过群集键访问数据的IN条件。当指定IN时，根据数据库版本，操作会出现在执行计划中。实际上，一直到11.1版本（包括该版本）使用的都是CONCATENATION操作，从11.2版本以后使用INLIST ITERATOR替代。这两个操作的每个子操作都执行一次来获取特定的群集键。下面的执行计划在11.1.0.7版本中生成：

```
SELECT * FROM t WHERE id IN (6, 8, 19, 28)

-------------------------------------------------------------
| Id | Operation           | Name | Starts | A-Rows | Buffers |
-------------------------------------------------------------
|  0 | SELECT STATEMENT    |      |      1 |      4 |       4 |
|  1 |  CONCATENATION      |      |      1 |      4 |       4 |
|* 2 |   TABLE ACCESS HASH| T    |      1 |      1 |       1 |
|* 3 |   TABLE ACCESS HASH| T    |      1 |      1 |       1 |
|* 4 |   TABLE ACCESS HASH| T    |      1 |      1 |       1 |
|* 5 |   TABLE ACCESS HASH| T    |      1 |      1 |       1 |
-------------------------------------------------------------

  2 - access("ID"=28)
  3 - access("ID"=19)
  4 - access("ID"=8)
  5 - access("ID"=6)
```

下面的执行计划在11.2.0.1版本中生成：

```
SELECT * FROM t WHERE id IN (6, 8, 19, 28)

-------------------------------------------------------------
| Id | Operation           | Name | Starts | A-Rows | Buffers |
-------------------------------------------------------------
|  0 | SELECT STATEMENT    |      |      1 |      4 |       4 |
|  1 |  INLIST ITERATOR    |      |      1 |      4 |       4 |
|* 2 |   TABLE ACCESS HASH| T    |      4 |      4 |       4 |
-------------------------------------------------------------

  2 - access(("ID"=6 OR "ID"=8 OR "ID"=19 OR "ID"=28))
```

重点需要强调的是，如果没有使用索引，其他所有条件都会导致全表扫描。例如，下面的查询，在WHERE子句中包含范围条件，使用索引：

```
SELECT * FROM t WHERE id < 6

-----------------------------------------------------------------------
| Id | Operation                    | Name | Starts | A-Rows | Buffers |
-----------------------------------------------------------------------
|  0 | SELECT STATEMENT             |      |      1 |      5 |       5 |
|  1 |  TABLE ACCESS BY INDEX ROWID| T    |      1 |      5 |       5 |
|* 2 |   INDEX RANGE SCAN           | T_PK |      1 |      5 |       3 |
-----------------------------------------------------------------------

  2 - access("ID"<6)
```

请注意，群集有特定的对象统计信息用来提供每个键的平均块数。可以通过user_clusters视图的avg_blocks_per_key列来显示（当然，还有dba、all和在12.1多租户环境下的cdb视图版本可用）。不幸的是，它的统计信息并不是由dbms_stats包收集的，反而需要你执行ANALYZE CLUSTER语句。为了更准确地估量值，别忘了收集它。

13.4 小结

本章不仅介绍了在选择高效访问路径时选择性的重要性，也介绍了用于访问在单表中存储的数据的不同方法。为此，弱选择性的SQL语句应该使用全表扫描、全分区扫描或全索引扫描。也讨论了为了高效执行强选择性的SQL语句，选择基于rowid、索引和单表散列群集的访问路径。

本章仅介绍了处理单表的SQL语句。实际上，多表联接很常见。为了解决这个问题，下一章将介绍三种基本联接方法及其利弊。换句话说，下章将介绍该在何时使用哪种联接方法。

第 14 章

优化联接

当一条SQL语句引用多张表时，查询优化器必须决定的事情，除了每张表的访问路径以外，还有表联接的顺序以及使用的联接方法。查询优化器的目标是尽可能早地过滤掉不必要的数据，以便最小化需要处理的数据总量。

本章首先会定义一些关键术语并解释三种基本的联接方法（嵌套循环、合并联接和散列联接）如何工作。接下来会给出一些如何选择联接方法的建议。最后，会介绍诸如分区智能联接和星型转换这样的优化技术。

注意　在本章中，多条SQL语句包含hint。这么做不仅是向你展示哪一个hint会导致哪一个执行计划，而且还为了向你展示它们的用法。无论如何，既没有提供真实的参考也没有提供完整的语法。可以在*Oracle Database SQL Reference*手册的第2章中找到相关信息。

14.1　定义

为避免误解，接下来的小节会定义一些贯穿本章的术语和概念。尤其是我会涉及不同类型的联接树、限制条件和联接条件之间的区别，以及不同类型的联接。

14.1.1　联接树

数据库引擎支持的所有联接方法在同一时刻都是只会处理两组数据。这两组数据被称为左输入和右输入。以这样的方式命名它们是因为当使用图示（见图14-1）时，输入的其中一个放置在联接的左边（T1）而另一个放置在右边（T2）。注意，在图示中，位于左边的节点要先于右边的节点执行。

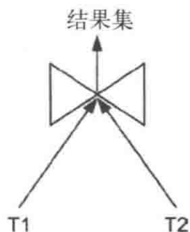


图14-1　两组数据之间联接的图示

当必须联接超过两组的数据时，查询优化器会评估联接树（Join Tree）。被查询优化器利用的联接树的类型会在接下来的四个小节中介绍。

1. 左深树

左深树（left-deep tree），如图14-2所示，是一种每个联接都有一张表（也就是说，不是由上一次联接生成的结果集）作为它的右输入的联接树。这是最经常被查询优化器选择的联接树。

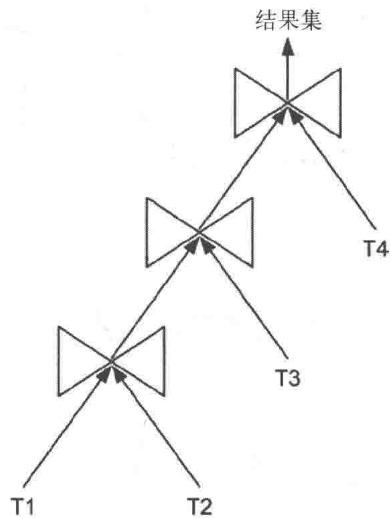


图14-2　在左深树中，右输入永远是一张表

下面的执行计划演示了图14-2描述的联接树。注意，每个联接操作（就是第5行、第6行和第7行）的第二个子操作（那就是，右输入）永远是一张表：

```
------------------------------------
| Id  | Operation          | Name |
------------------------------------
|   0 | SELECT STATEMENT   |      |
|   1 |  HASH JOIN         |      |
|   2 |   HASH JOIN        |      |
|   3 |    HASH JOIN       |      |
|   4 |     TABLE ACCESS FULL| T1  |
|   5 |     TABLE ACCESS FULL| T2  |
|   6 |    TABLE ACCESS FULL | T3  |
|   7 |   TABLE ACCESS FULL  | T4  |
------------------------------------
```

2. 右深树

右深树（right-deep tree），如图14-3所示，是一种每个联接都有一张表作为它的左输入的联接树。这个联接树很少会被查询优化器选择。

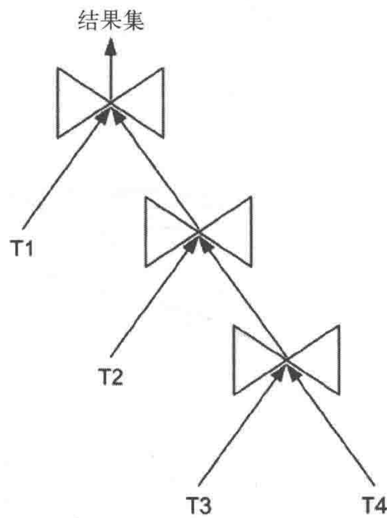


图14-3　在右深树中，左输入永远是一张表

下面的执行计划演示了图14-3描述的联接树。注意，每个联接操作（就是第2行、第4行和第6行）的第一个子操作（那就是，左输入）永远是一张表：

```
------------------------------------------
| Id  | Operation               | Name |
------------------------------------------
|   0 | SELECT STATEMENT        |      |
|   1 |  HASH JOIN              |      |
|   2 |   TABLE ACCESS FULL     | T1   |
|   3 |   HASH JOIN             |      |
|   4 |    TABLE ACCESS FULL    | T2   |
|   5 |    HASH JOIN            |      |
|   6 |     TABLE ACCESS FULL| T3   |
|   7 |     TABLE ACCESS FULL| T4   |
------------------------------------------
```

3. 曲折树

曲折树（zig-zag tree），如图14-4所示，这种联接树的每个联接都至少有一张表作为输入，但是这些基于表的输入有时候位于左侧有时候位于右侧。这种类型的联接树通常不会被查询优化器所使用。

下面的执行计划演示了图14-4描述的联接树：

```
------------------------------------------
| Id  | Operation               | Name |
------------------------------------------
|   0 | SELECT STATEMENT        |      |
|   1 |  HASH JOIN              |      |
|   2 |   HASH JOIN             |      |
|   3 |    TABLE ACCESS FULL    | T1   |
|   4 |    HASH JOIN            |      |
|   5 |     TABLE ACCESS FULL| T2   |
```

```
|   6 |     TABLE ACCESS FULL| T3   |
|   7 |   TABLE ACCESS FULL  | T4   |
-------------------------------------
```

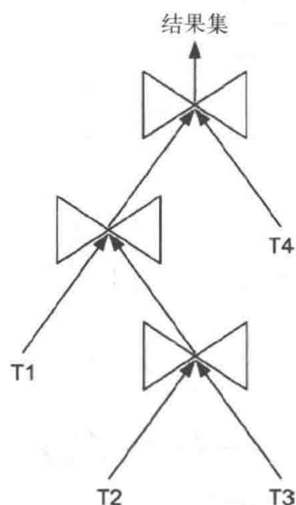


图14-4　在曲折树中，两个输入中至少有一个是一张表

4. 浓密树

浓密树（bushy tree），如图14-5所示，是一个可能拥有两个输入都不是表的联接的联接树。换句话说，这种树的结构是完全自由的。查询优化器只会在没有其他选项可用时才会选择这种类型的联接树。通常在不可合并的视图或子查询出现的时候会发生这种情况。

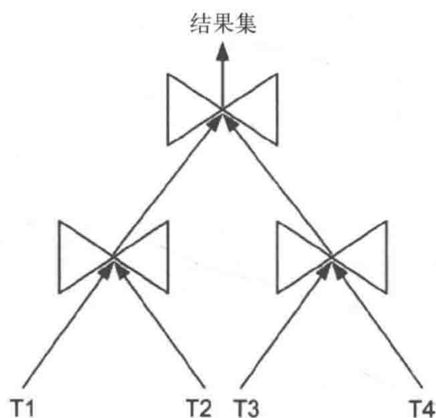


图14-5　浓密树的结构是完全自由的

下面的执行计划演示了图14-5描述的联接树。注意，联接操作1的子操作是另外两个联接操作的结果集：

```
-------------------------------------------
| Id  | Operation              | Name |
-------------------------------------------
|   0 | SELECT STATEMENT       |      |
|   1 |  HASH JOIN             |      |
|   2 |   VIEW                 |      |
|   3 |    HASH JOIN           |      |
|   4 |     TABLE ACCESS FULL| T1   |
|   5 |     TABLE ACCESS FULL| T2   |
|   6 |   VIEW                 |      |
|   7 |    HASH JOIN           |      |
|   8 |     TABLE ACCESS FULL| T3   |
|   9 |     TABLE ACCESS FULL| T4   |
-------------------------------------------
```

14.1.2 联接的类型

有两种类型用于指定联接的语法。经典语法，是在很早的SQL标准（SQL-86）中指定的，使用FROM子句和WHERE子句两者一起来指定联接。较新的语法，第一次是在SQL-92中可用，仅使用FROM子句来指定一个联接。新的语法有时称作*ANSI联接语法*。不管怎样，从SQL标准的视角来看两种语法类型都是有效的。在Oracle数据库中，因为历史原因，使用最频繁的语法是经典的那种。事实上，不仅很多的开发人员和DBA习惯这种语法，而且很多应用程序也是使用它开发的。虽然如此，新的语法提供经典语法不支持的选项。接下来的小节会基于两种语法提供案例。这里使用的所有查询都作为例子在join_types.sql脚本中提供。

注意 这一节中描述的联接类型并非互相排斥的。一个给定的联接可能会属于不止一种类别。例如，认为一个内联接也是一个自联接是完全可信的。

---

1. 交叉联接

交叉联接（cross join），也称作*笛卡儿积*（Cartesian product），是将一张表的每一行与另外一张表的每一行组合的操作。这种类型的操作会在下面的查询中列举的两种情况下出现。第一个使用经典联接语法（没有指定联接条件）：

```
SELECT emp.ename, dept.dname FROM emp, dept
```

第二个使用新的联接语法（使用了CROSS JOIN）：

```
SELECT emp.ename, dept.dname FROM emp CROSS JOIN dept
```

在现实中，交叉联接几乎是不需要的。虽然如此，后者的语法能更好地证明开发者的意图。事实上，明确指定是有好处的。使用前者，则不清楚是否是写下该SQL语句的那个人忘记了WHERE子句。

2. θ联接

θ联接（theta join）等同于在一个交叉联接的结果集上执行一个选择。换句话说，取代返回一张表的每一行与另一张表的每一行的组合，只有满足联接条件的记录会被返回。下面的两个查询是这种

类型联接的例子：

```
SELECT emp.ename, salgrade.grade
FROM emp, salgrade
WHERE emp.sal BETWEEN salgrade.losal AND salgrade.hisal

SELECT emp.ename, salgrade.grade
FROM emp JOIN salgrade ON emp.sal BETWEEN salgrade.losal AND salgrade.hisal
```

θ联接也被称作内联接（inner join）。在上面的查询中使用的是新的联接语法，关键字INNER被省略了，但其实可以对其进行显式编码，如下例所示：

```
SELECT emp.ename, salgrade.grade
FROM emp INNER JOIN salgrade ON emp.sal BETWEEN salgrade.losal AND salgrade.hisal
```

3. 等值联接

等值联接（equi-join）是一种只在联接条件中使用等值运算符的特殊类型的内联接。下面的两个查询是例子：

```
SELECT emp.ename, dept.dname
FROM emp, dept
WHERE emp.deptno = dept.deptno

SELECT emp.ename, dept.dname
FROM emp JOIN dept ON emp.deptno = dept.deptno
```

4. 自联接

自联接（self-join）一种一张表联接自己的特殊类型的内联接。下面的两个查询是这种联接的例子。注意，emp表在FROM子句中被引用了两次：

```
SELECT emp.ename, mgr.ename
FROM emp, emp mgr
WHERE emp.mgr = mgr.empno

SELECT emp.ename, mgr.ename
FROM emp JOIN emp mgr ON emp.mgr = mgr.empno
```

5. 外联接

外联接（outer join）扩展内联接的结果集。事实上，通过一个外联接，即使在另外一张表中没有找到匹配的值也会返回一张表（保留表）的所有记录。NULL值会与那张不包含任何匹配数据的表返回的列进行关联。举个例子，在上一部分（自联接）中的查询不会返回emp表的所有数据，因为雇员KING，也就是主席，没有管理者。要使用经典语法指定外联接，必须使用一个Oracle的扩展（基于运算符(+)）。下面的查询是一个例子：

```
SELECT emp.ename, mgr.ename
FROM emp, emp mgr
WHERE emp.mgr = mgr.empno(+)
```

要使用新的语法指定外联接，存在几种可行性。举例来说，下面的两个查询等同于上一个：

```
SELECT emp.ename, mgr.ename
FROM emp LEFT JOIN emp mgr ON emp.mgr = mgr.empno

SELECT emp.ename, mgr.ename
FROM emp mgr RIGHT JOIN emp ON emp.mgr = mgr.empno
```

下面的查询展示，对于内联接，可能会添加OUTER关键字，以显式指定这是一个外联接：

```
SELECT emp.ename, mgr.ename
FROM emp LEFT OUTER JOIN emp mgr ON emp.mgr = mgr.empno
```

此外，使用新的联接语法，可以借助于完全外联接（full outer join）指定返回两张表的全部数据。换句话说，两张表中在另外一张表中没有匹配记录的所有数据都要保留。下面的查询是一个例子：

```
SELECT mgr.ename AS manager, emp.ename AS subordinate
FROM emp FULL OUTER JOIN emp mgr ON emp.mgr = mgr.empno
```

另一种可能性是指定一个已分区外联接（partitioned outer join）。注意：此处分区的意思与第13章中讨论的对象物理分区没有任何关系。相反，它的意思是数据在运行时被分成多个子集。其思路是不在两张表之间执行外联接，而是在一张表与另外一张表的子集之间执行。举例来说，在下面的查询中，emp表被基于job列分成多个子集。然后每个子集与dept表进行外联接：

```
SELECT dept.dname, count(emp.empno)
FROM dept LEFT JOIN emp PARTITION BY (emp.job) ON emp.deptno = dept.deptno
WHERE emp.job = 'MANAGER'
GROUP BY dept.dname
```

6. 半联接

两张表之间的半联接（semi-join）会在另外一张表中找到匹配的记录时，返回当前这张表的数据。与自联接相反，来自左输入的数据至多被返回一次。此外，来自右输入的数据根本不会被返回。联接条件是通过IN、EXISTS、ANY或SOME编写的。下面的查询是一个例子：

```
SELECT deptno, dname, loc
FROM dept
WHERE deptno IN (SELECT deptno FROM emp)

SELECT deptno, dname, loc
FROM dept
WHERE EXISTS (SELECT deptno FROM emp WHERE emp.deptno = dept.deptno)

SELECT deptno, dname, loc
FROM dept
WHERE deptno = ANY (SELECT deptno FROM emp)

SELECT deptno, dname, loc
FROM dept
WHERE deptno = SOME (SELECT deptno FROM emp)
```


7. 反联接

反联接（anti-join）是一种特殊类型的半联接，只有在另外一张表中没有匹配记录的那些数据会被返回。联接条件通常是使用NOT IN或NOT EXISTS编写的。下面的两个查询是例子：

```
SELECT deptno, dname, loc
FROM dept
WHERE deptno NOT IN (SELECT deptno FROM emp)

SELECT deptno, dname, loc
FROM dept
WHERE NOT EXISTS (SELECT deptno FROM emp WHERE emp.deptno = dept.deptno)
```

8. 横向内联视图

横向内联视图（lateral inline view）是一个内联视图（inline view，在另一个查询的FROM子句中指定的查询），包含着指向FROM子句中先于它出现的其他表的关联条件。从12.1版本开始，横向内联视图是通过LATERAL关键字来支持的。下面的查询展示了一个例子：

```
SELECT dname, ename
FROM dept, LATERAL(SELECT * FROM emp WHERE dept.deptno = emp.deptno)
```

注意，如果缺少LATERAL 关键字，会引发以下错误：

```
SQL> SELECT dname, empno
  2  FROM dept, (SELECT * FROM emp WHERE dept.deptno = emp.deptno);
FROM dept, (SELECT * FROM emp WHERE dept.deptno = emp.deptno)
                                    *
ERROR at line 2:
ORA-00904: "DEPT"."DEPTNO": invalid identifier
```

对于外联接和交叉联接，通过OUTER APPLY和CROSS APPLY关键字提供了类似的功能。

14.1.3 限制条件与联接条件

为选择一个联接方法，理解限制条件（也称为过滤条件）与联接条件之间的区别非常重要。从语法的角度看，仅当使用经典联接语法时，这两者才会造成困惑。事实上，使用经典联接语法，WHERE子句同时被用来指定限制条件和联接条件。使用新的联接语法，限制条件在WHERE子句中指定，而联接条件则在FROM子句中指定。下面的伪SQL语句演示了此语法：

```
SELECT <columns>
FROM <table1> [OUTER] JOIN <table2> ON ( <join conditions> )
WHERE <restrictions>
```

从概念的角度看，一条包含联接条件和限制条件的SQL语句以下面的方式执行。

- 这两组数据基于联接条件联接。
- 限制条件被应用于联接返回的结果集。

换句话说，在联接两组数据的时候，一个联接条件被指定以避免交叉联接。它并不打算过滤掉结果集。反而，会指定一个限制条件，以过滤由上一个操作（例如，一个联接）返回的结果集。举例来说，下面的查询，联接条件是emp.deptno = dept.deptno，而限制条件是dept.loc = 'DALLAS'：

```
SELECT emp.ename
FROM emp, dept
WHERE emp.deptno = dept.deptno
AND dept.loc = 'DALLAS'
```

从实现的角度看，查询优化器同时利用限制条件和联接条件没什么不寻常的。一方面，联接条件可能被用于过滤掉数据。另一方面，限制条件可能会在联接条件之前被评估以最小化需要联接的数据总量。举例来说，上面的查询可能会使用以下的执行计划执行。注意dept.loc = 'DALLAS'这个限制条件（操作2）是如何早于emp.deptno = dept.deptno这个联接条件（操作1）被应用的：

```
-----------------------------------
| Id  | Operation           | Name |
-----------------------------------
|   0 | SELECT STATEMENT    |      |
|*  1 |  HASH JOIN          |      |
|*  2 |   TABLE ACCESS FULL| DEPT |
|   3 |   TABLE ACCESS FULL| EMP  |
-----------------------------------

   1 - access("EMP"."DEPTNO"="DEPT"."DEPTNO")
   2 - filter("DEPT"."LOC"='DALLAS')
```

14.2 嵌套循环联接

接下来的小节会介绍*嵌套循环联接*（nested loop join）是如何工作的。我会描述它们的普遍行为，然后会给出几个两表联接和四表联接的例子。最后，我会介绍一些优化技巧。所有的例子都来自nested_loops_join.sql这个脚本。

14.2.1 概念

由嵌套循环联接处理的两组数据称作*外循环*（也称作*驱动行源*）和*内循环*。外循环是左输入，而内循环则是右输入。如图14-6所列举的那样，外循环执行一次，内循环则为由外循环返回的每一行数据都执行一次。

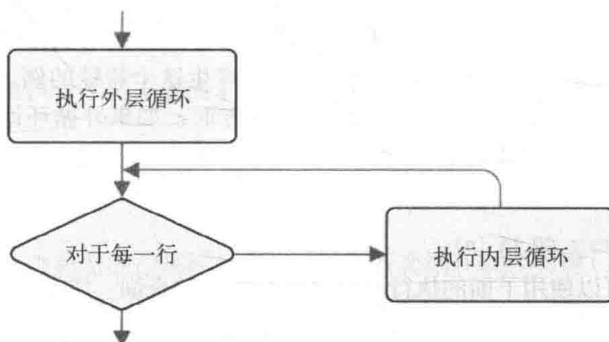


图14-6 概览由嵌套循环联接执行的处理

嵌套循环联接拥有的具体特征如下所示。

- 左输入（外循环）只执行一次。右输入（内循环）可能会执行很多次。
- 它们能够在处理完所有数据之前就返回结果集的第一行。

- ❑ 它们既可以利用索引应用于限制条件上，也可以应用于联接条件上。
- ❑ 它们支持所有类型的联接。

14.2.2 两表联接

下面是一个处理两表之间的嵌套循环联接的样例执行计划。该样例还展示如何通过使用leading和use_nl hint强制执行一个嵌套循环联接。前者表明表被访问的顺序。换句话说，它指定哪张表是在外循环（t1）中访问以及哪张表是在内循环（t2）中访问。后者指定哪种联接方法用于联接内循环返回的数据和t1表。一定要注意，use_nl hint不包含对表t1的引用：

```
SELECT /*+ leading(t1 t2) use_nl(t2) full(t1) full(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 19

-----------------------------------
| Id  | Operation           | Name |
-----------------------------------
|   0 | SELECT STATEMENT    |      |
|   1 |  NESTED LOOPS       |      |
|*  2 |   TABLE ACCESS FULL| T1   |
|*  3 |   TABLE ACCESS FULL| T2   |
-----------------------------------

   2 - filter("T1"."N"=19)
   3 - filter("T1"."ID"="T2"."T1_ID")
```

如第10章中所述，NESTED LOOPS操作属于关联组合类型。这意味着第一个子操作（外循环）控制着第二个子操作（内循环）的执行。在此案例中，执行计划的处理过程可以总结成以下这样。

- ❑ 表t1中的所有数据都通过一次全表扫描读取，接下来应用了n = 19限制条件。
- ❑ 表t2的全表扫描执行的次数与上一步中返回的行数相同。

显然，当操作2（TABLE ACCESS FULL）返回超过一行时，上面的执行计划不是高效的，因此，几乎永远不会被查询优化器选择。基于这个原因，为了产生这个特殊的例子，有必要指定两个访问hint（full）来强制查询优化器使用此执行计划。另一方面，如果外循环返回一个单独的行且内循环的选择率很弱，表t2的全表扫描可能就是合理的。为了证实，我们为表t1的列n创建了下面的唯一索引：

```
CREATE UNIQUE INDEX t1_n ON t1 (n)
```

有了这个索引，就可以使用下面的执行计划来执行前面的查询。注意，这一次只有一个控制t2表的访问路径的hint被指定。其他的hint没有必要，因为查询优化器清楚在这样的环境下嵌套循环联接是执行此查询最高效的方式。事实上，因为操作3（INDEX UNIQUE SCAN）的缘故，可以确保内循环仅执行一次：

```
SELECT /*+ full(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 19
```

```
-------------------------------------------------
| Id  | Operation                    | Name  |
-------------------------------------------------
|   0 | SELECT STATEMENT             |       |
|   1 |  NESTED LOOPS                |       |
|   2 |   TABLE ACCESS BY INDEX ROWID| T1    |
|*  3 |    INDEX UNIQUE SCAN         | T1_N  |
|*  4 |   TABLE ACCESS FULL          | T2    |
-------------------------------------------------

   3 - access("T1"."N"=19)
   4 - filter("T1"."ID"="T2"."T1_ID")
```

就像在上一小节中讨论过的那样，如果内循环的选择性很强，为内循环使用索引扫描是合理的。因为嵌套循环联接是一个关联组合操作，对于内循环来说甚至可能利用联接条件来实现索引扫描。举例来说，在下面的执行计划中，操作5使用操作3返回的列t1.id的值进行了检索：

```
SELECT /*+ ordered use_nl(t2) index(t1) index(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 19

-----------------------------------------------------
| Id  | Operation                    | Name      |
-----------------------------------------------------
|   0 | SELECT STATEMENT             |           |
|   1 |  NESTED LOOPS                |           |
|   2 |   TABLE ACCESS BY INDEX ROWID| T1        |
|*  3 |    INDEX UNIQUE SCAN         | T1_N      |
|   4 |   TABLE ACCESS BY INDEX ROWID| T2        |
|*  5 |    INDEX RANGE SCAN          | T2_T1_ID  |
-----------------------------------------------------

   3 - access("T1"."N"=19)
   5 - access("T1"."ID"="T2"."T1_ID")
```

总之，如果内循环被执行了几（或很多）次，只有假设具有强选择性以及只会导致很少的逻辑读的访问路径是合理的。

14.2.3 四表联接

下面的执行计划是一个典型的左深树的例子，使用嵌套循环联接实现（图示请参考图14-2）。注意每张表都是如何通过索引访问的。这个例子还展示了如何通过ordered 和use_nl hint来强制执行嵌套循环联接。前者指定按照表在FROM子句中出现的相同顺序来访问它们。后者指定哪种联接方法用于联接由hint引用的表和第一张表或上一个联接操作的结果集：

```
SELECT /*+ ordered use_nl(t2 t3 t4) */ t1.*, t2.*, t3.*, t4.*
FROM t1, t2, t3, t4
WHERE t1.id = t2.t1_id
AND t2.id = t3.t2_id
AND t3.id = t4.t3_id
```

```
AND t1.n = 19

------------------------------------------------------
| Id  | Operation                     | Name       |
------------------------------------------------------
|   0 | SELECT STATEMENT              |            |
|   1 |  NESTED LOOPS                 |            |
|   2 |   NESTED LOOPS                |            |
|   3 |    NESTED LOOPS               |            |
|   4 |     TABLE ACCESS BY INDEX ROWID| T1        |
|*  5 |      INDEX RANGE SCAN         | T1_N       |
|   6 |     TABLE ACCESS BY INDEX ROWID| T2        |
|*  7 |      INDEX RANGE SCAN         | T2_T1_ID   |
|   8 |    TABLE ACCESS BY INDEX ROWID | T3        |
|*  9 |     INDEX RANGE SCAN          | T3_T2_ID   |
|  10 |    TABLE ACCESS BY INDEX ROWID | T4        |
|* 11 |     INDEX RANGE SCAN          | T4_T3_ID   |
------------------------------------------------------

  5 - access("T1"."N"=19)
  7 - access("T1"."ID"="T2"."T1_ID")
  9 - access("T2"."ID"="T3"."T2_ID")
 11 - access("T3"."ID"="T4"."T3_ID")
```

这种类型的执行计划的处理过程可以总结成如下几点（该描述假设没有使用行预取）。

(1) 当提取第一行（换句话说，不是在查询被解析或执行的时候）时，处理过程以从表t1中获取满足t1.n = 19限制条件的第一行为开始。

(2) 根据在表t1中找到的数据，检索表t2。注意，数据库引擎利用t1.id = t2.t1_id联接条件访问表t2。事实上，那张表上没有应用任何限制条件。只有第一条满足联接条件的记录被返回给父操作。

(3) 根据在表t2中找到的数据，检索t3表。在这个情况中也一样，数据库引擎利用联接条件t2.id = t3.t2_id来访问表t3。只有第一条满足联接条件的记录被返回给父操作。

(4) 根据在表t3中找到的数据，检索t4表。这里也是一样，数据库引擎利用联接条件t3.id = t4.t3_id来访问表t4。第一条满足联接条件的记录被立即返回给客户端。

(5) 当提取随后的记录后，会执行与第一次提取时一样的动作。显然，处理过程从紧邻上一次匹配的位置重新开始（可以是匹配表t4的第二条记录，如果存在的话）。这里有必要强调一下，一旦找到第一条满足要求的记录，就会尽快返回该记录。没有必要在返回第一条数据前完全执行联接。

14.2.4　缓冲区缓存预取

基本上，每条访问路径，除了全扫描，都会导致缓存未命中事件中的单块物理读。对于嵌套循环联接，尤其是需要处理大量数据的时候，这些单块物理读的效率可能非常低下。事实上，对于嵌套循环联接来说，伴随很多的单块物理读来访问数据块也没有什么不寻常的。

为了改进嵌套循环联接的效率，数据库引擎能够利用以多块物理读代替单块物理读的优化技术。有三种特性使用这样的方法：表预取（table prefetching）、批处理（batching）以及缓冲区缓存预热（buffer cache prewarm）。前两个与本节呈现的执行计划有关；最后一个只会在实例重启后因为嵌套循环联接而短暂出现。注意这些优化技术在访问表和索引的时候都会执行。

14.2.2节展示了一个具有以下形状的执行计划，该执行计划基于嵌套循环联接：

```
-----------------------------------------------------------
| Id  | Operation                     | Name     |
-----------------------------------------------------------
|   0 | SELECT STATEMENT              |          |
|   1 |  NESTED LOOPS                 |          |
|   2 |   TABLE ACCESS BY INDEX ROWID| T1       |
|*  3 |    INDEX UNIQUE SCAN          | T1_N     |
|   4 |   TABLE ACCESS BY INDEX ROWID| T2       |
|*  5 |    INDEX RANGE SCAN           | T2_T1_ID |
-----------------------------------------------------------

   3 - access("T1"."N"=19)
   5 - access("T1"."ID"="T2"."T1_ID")
```

在实践中，对于本书涵盖的Oracle数据库版本来说，这种类型的执行计划只用于基于索引的唯一扫描（此处t1_n索引是唯一的）的外循环或内循环。我们来看一下如果列n上的t1_n索引按以下方式（非唯一）定义会发生什么：

```
CREATE INDEX t1_n ON t1 (n)
```

有了这个索引，就会使用以下执行计划。注意表t2上的rowid访问的不同位置。在上一个计划中，它是操作4，而在接下来的这个计划中，它是操作1。非常独特的是，rowid访问（操作1）的子操作是嵌套循环联接（操作2）。即使从功能的角度来看这两个执行计划是等效的，但数据库引擎使用具有以下形状的执行计划以利用表预取：

```
------------------------------------------------------------
| Id  | Operation                      | Name     |
------------------------------------------------------------
|   0 | SELECT STATEMENT               |          |
|   1 |  TABLE ACCESS BY INDEX ROWID   | T2       |
|   2 |   NESTED LOOPS                 |          |
|   3 |    TABLE ACCESS BY INDEX ROWID| T1       |
|*  4 |     INDEX RANGE SCAN           | T1_N     |
|*  5 |    INDEX RANGE SCAN            | T2_T1_ID |
------------------------------------------------------------

   4 - access("T1"."N"=19)
   5 - access("T1"."ID"="T2"."T1_ID")
```

从11.1版本开始，可以通过nlj_prefetch和no_nlj_prefetch hint来控制上面执行计划的使用。

从11.1版本开始，为进一步优化嵌套循环联接，表预取被批处理所取代。结果，应该可以观察到下面的执行计划而非上面的那个：

```
------------------------------------------------------------
| Id  | Operation                      | Name     |
------------------------------------------------------------
|   0 | SELECT STATEMENT               |          |
|   1 |  NESTED LOOPS                  |          |
|   2 |   NESTED LOOPS                 |          |
|   3 |    TABLE ACCESS BY INDEX ROWID| T1       |
|*  4 |     INDEX RANGE SCAN           | T1_N     |
|*  5 |    INDEX RANGE SCAN            | T2_T1_ID |
```

```
|   6 |    TABLE ACCESS BY INDEX ROWID | T2         |
---------------------------------------------------

   4 - access("T1"."N"=19)
   5 - access("T1"."ID"="T2"."T1_ID")
```

注意，尽管查询总是相同的（就是说，两表联接），但是该执行计划却包含两个嵌套循环联接！要控制批处理，可以使用nlj_batching 和no_nlj_batching hint。

查看执行计划并不能告诉你数据库引擎是否使用了表预取或批处理。事实是，尽管是查询优化器生成一个可以利用表预取或批处理的执行计划，但却是执行引擎决定使用这个计划是否合理。了解一项优化技术是否被使用的唯一方式是查看服务进程执行的物理读，尤其是与其有关的等待事件。

- db file sequential read事件与单块物理读有关。因此，如果此事件出现，那么要么是没有使用优化技术，要么是不需要优化技术（例如，因为所需的数据块已经在缓冲区缓存中）。
- db file scattered read和db file parallel read事件与多块物理读有关。这两个事件之间的不同在于前者是用于相邻数据块的物理读，而后者是用于非相邻块的物理读。因此，如果它们其中的一个出现在rowid访问或索引范围扫描中，说明没有使用任何优化技术。

14.3　合并联接

接下来的小节描述合并联接（merge join，也称作排序合并联接，sort-merge join）是如何工作的。我会描述它们的共性行为，并给出一些两表联接和四表联接的例子作为开头。最后，我会描述在处理期间使用的工作区。所有的例子都来自merge_join.sql脚本。

14.3.1　概念

取决于SQL语句和物理数据库设计，在执行合并联接时数据库引擎可以在多种方式之间进行选择。在一般情况下，两组数据都会根据联接条件的列进行读取和排序。一旦这些操作执行完毕，包含在两个工作区中的数据就会被合并，如图14-7所示。

合并联接拥有的具体特征如下所示。

- 左输入仅被执行一次。
- 右输入至多被执行一次。如果左输入不返回任何数据，右输入则根本不会被执行。
- 除了执行了笛卡儿积的情况，两个输入返回的数据必须被根据联接条件的列进行排序。
- 当数据进行排序后，在返回结果集的第一行数据之前，必须完整读取并排序两个输入。
- 支持所有类型的联接。

合并联接的使用并不是很频繁。原因是，在大多数的情况下，无论是嵌套循环联接还是散列联接，都执行得比合并联接要更好。

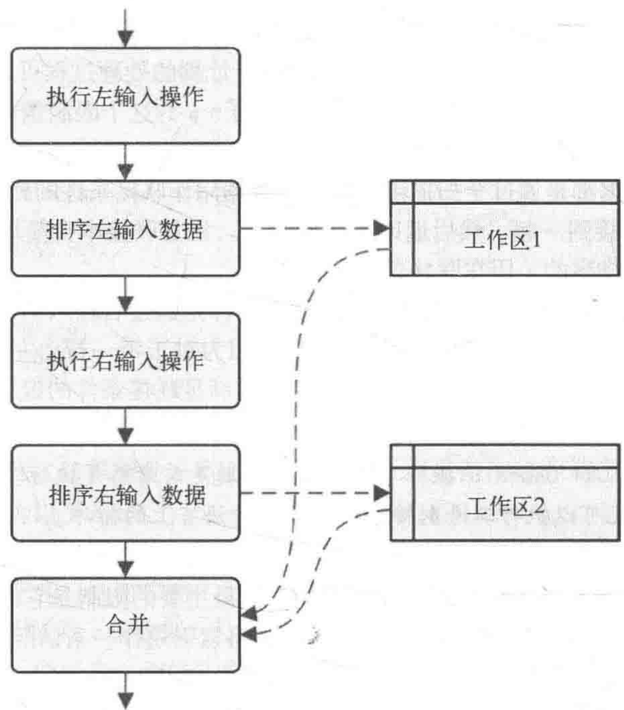


图14-7 合并联接执行的处理过程概览

14.3.2 两表联接

下面是一个简单的处理两表之间合并联接的执行计划。该例子还展示了如何通过使用ordered和use_merge这两个hint来强制执行合并联接：

```
SELECT /*+ ordered use_merge(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 19

-----------------------------------
| Id  | Operation            | Name |
-----------------------------------
|   0 | SELECT STATEMENT     |      |
|   1 |  MERGE JOIN          |      |
|   2 |   SORT JOIN          |      |
|*  3 |    TABLE ACCESS FULL| T1   |
|*  4 |   SORT JOIN          |      |
|   5 |    TABLE ACCESS FULL| T2   |
-----------------------------------

   3 - filter("T1"."N"=19)
   4 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")
```

如第10章中所描述的，MERGE JOIN操作是非关联组合的类型。这意味着两个子操作至多被处理一次并且互相独立。假设两个输入都返回数据，上面的执行计划的处理过程可以总结为以下几点。

- 表t1中的所有数据都是通过全扫描读取的，应用了n = 19这个限制条件，而且结果数据根据用作联接条件的列（id）进行了排序。
- 表t2中的所有数据都是通过全扫描读取的，并根据用作联接条件的列（t1_id）进行了排序。
- 这两组数据被联接到一起，然后返回了结果数据。注意联接本身简单明了，因为这两组数据是根据相同的值排序的（用在联接条件中的列）。

有趣的是，我们留意到在上面的执行计划中，联接条件是通过右输入的SORT JOIN操作应用的，而不是通过也许你会预期的MERGE JOIN操作应用的。这是因为对于每一行由左输入返回的数据，MERGE JOIN操作都会访问与右输入有关的内存结构来检查是否有满足联接条件的记录存在。

注意　SORT JOIN操作与SORT ORDER BY操作之间的关键区别是前者总是执行二进制排序，而后者，取决于NLS配置，既可以执行二进制排序也可以执行语言上的排序。

MERGE JOIN操作（对于其他无关联组合操作也类似）最主要的限制是它没有能力通过在联接条件上应用索引来获益。换句话说，仅可以在对输入排序之前将索引用作一条访问路径来验证限制条件（如果已指定）。因此，为了选择访问路径，你不得不为两张表应用在第13章中讨论的方法。举个例子，如果n=19这个限制条件提供强选择性，在此列上创建索引并应用，效果会很明显：

```
CREATE INDEX t1_n ON t1 (n)
```

有了这个索引后，可能会使用以下执行计划。你应该注意到，不再通过全表扫描访问表t1了：

```
---------------------------------------------
| Id  | Operation                    | Name |
---------------------------------------------
|   0 | SELECT STATEMENT             |      |
|   1 |  MERGE JOIN                  |      |
|   2 |   SORT JOIN                  |      |
|   3 |    TABLE ACCESS BY INDEX ROWID| T1   |
|*  4 |     INDEX RANGE SCAN         | T1_N |
|*  5 |   SORT JOIN                  |      |
|   6 |    TABLE ACCESS FULL         | T2   |
---------------------------------------------

   4 - access("T1"."N"=19)
   5 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")
```

要执行合并联接，可能会有不可忽视的资源总量消耗在排序上。为改进性能，只要能够节省资源，查询优化器无论何时都会避免执行排序操作。但只有当数据已根据用作联接条件的列进行了排序时这才可行。这种行为会在两种情况下出现。第一种是索引范围扫描利用在作为联接条件使用的列上创建的索引时。第二种是在合并排序的上一步（例如，另一个排序合并）中已经将数据以正确的顺序进行了排序时。举例来说，在下面的执行计划中，注意表t1是如何通过t1_pk索引（也就是在用作联接条件的id列上创建的索引）进行访问的。因此，对于左输入，就避免了排序操作（SORT JOIN）：

```
-----------------------------------------------
| Id  | Operation                    | Name   |
-----------------------------------------------
|   0 | SELECT STATEMENT             |        |
|   1 |  MERGE JOIN                  |        |
|*  2 |   TABLE ACCESS BY INDEX ROWID| T1     |
|   3 |    INDEX FULL SCAN           | T1_PK  |
|*  4 |   SORT JOIN                  |        |
|   5 |    TABLE ACCESS FULL         | T2     |
-----------------------------------------------

   2 - filter("T1"."N"=19)
   4 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")
```

一个关于类似上面这样的执行计划的重要警告是，因为对于左输入来讲没有发生任何排序，所以没有工作区与其进行关联。结果就是，当右输入执行时，没有空间来存储来自左输入的结果数据。上面的执行计划可以总结为以下几步。

(1) 第一批数据通过t1_pk索引从t1表中抽取出来。一定要理解第一批数据仅在结果集很小的时候才会包含所有的数据。记住，此时没有工作区来临时存储很多数据。

(2) 假设上一步中返回了一些数据，t2表中的所有数据都通过全表扫描读取出来，根据作为联接条件使用的列进行排序，并且在工作区中排序时，可能会将临时数据溢出到磁盘上。

(3) 将两组数据联接到一起，然后返回结果数据。当第一批抽取自左输入的数据已被完全处理时，继续从t1表抽取更多的数据，如果有必要，则与右输入已经呈现在工作区中的数据进行联接。

如上面的执行计划中所示，有可能避免与左输入相关的排序。但是，相同的情况却不适用于右输入。为解释原因，下面的例子展示在执行与以前一样的查询时让查询优化器选择执行计划，但是这一次通过leading这个hint指定，使得表t2必须在左输入中进行访问。注意，尽管右输入的访问路径按正确的顺序返回数据，但是数据还是必须经历一次SORT JOIN操作(4)：

```
SELECT /*+ leading(t2 t1) use_merge(t1) index(t1 t1_pk) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 19

------------------------------------------------
| Id  | Operation                     | Name   |
------------------------------------------------
|   0 | SELECT STATEMENT              |        |
|   1 |  MERGE JOIN                   |        |
|   2 |   SORT JOIN                   |        |
|   3 |    TABLE ACCESS FULL          | T2     |
|*  4 |   SORT JOIN                   |        |
|*  5 |    TABLE ACCESS BY INDEX ROWID| T1     |
|   6 |     INDEX FULL SCAN           | T1_PK  |
------------------------------------------------

   4 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")
   5 - filter("T1"."N"=19)
```

需要该SORT JOIN操作的原因是MERGE JOIN操作需要访问与右输入关联的内存结构，以便检查是否

有满足联接条件的数据存在。而这个访问必须在这样一个内存结构中执行，它不仅包含按指定顺序存储的数据，而且还要允许基于联接条件执行高效的检索。

基于合并联接的笛卡儿积是按照不同方式执行的。对比本节描述的所有其他案例，应用于它们的主要优化是数据不需要进行排序。原因很简单：没有联接条件存在，因此不可能根据不存在的联接条件中引用的列对数据进行排序。因此，与用于获取数据的访问路径无关，不需要SORT JOIN操作。对于右输入，不管怎样还是有必要在一个内存结构中存储数据。出于这个目的，会使用BUFFER SORT操作（不要看它的名称，它不会对数据进行排序）。下面的例子演示这样的一种情况：

```
SELECT /*+ ordered use_merge(t2) */ *
FROM t1, t2

-------------------------------------
| Id  | Operation             | Name |
-------------------------------------
|   0 | SELECT STATEMENT      |      |
|   1 |  MERGE JOIN CARTESIAN|      |
|   2 |   TABLE ACCESS FULL   | T1   |
|   3 |   BUFFER SORT         |      |
|   4 |    TABLE ACCESS FULL  | T2   |
-------------------------------------
```

14.3.3　四表联接

下面的执行计划是一个典型的通过合并联接实现左深树（图示请参见图14-2）的例子。该例子还展示如何依靠leading和use_merge这两个hint来强制执行合并联接。注意，leading这个hint支持多张表：

```
SELECT /*+ leading(t1 t2 t3 t4) use_merge(t2 t3 t4) */ t1.*, t2.*, t3.*, t4.*
FROM t1, t2, t3, t4
WHERE t1.id = t2.t1_id
AND t2.id = t3.t2_id
AND t3.id = t4.t3_id
AND t1.n = 19

----------------------------------------
| Id  | Operation              | Name |
----------------------------------------
|   0 | SELECT STATEMENT       |      |
|   1 |  MERGE JOIN            |      |
|   2 |   SORT JOIN            |      |
|   3 |    MERGE JOIN          |      |
|   4 |     SORT JOIN          |      |
|   5 |      MERGE JOIN        |      |
|   6 |       SORT JOIN        |      |
|*  7 |        TABLE ACCESS FULL| T1   |
|*  8 |       SORT JOIN        |      |
|   9 |        TABLE ACCESS FULL| T2   |
|* 10 |     SORT JOIN          |      |
|  11 |      TABLE ACCESS FULL  | T3   |
|* 12 |   SORT JOIN            |      |
|  13 |    TABLE ACCESS FULL   | T4   |
```

```
--------------------------------------

   7 - filter("T1"."N"=19)
   8 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")
  10 - access("T2"."ID"="T3"."T2_ID")
       filter("T2"."ID"="T3"."T2_ID")
  12 - access("T3"."ID"="T4"."T3_ID")
       filter("T3"."ID"="T4"."T3_ID")
```

该处理过程与上一小节中讨论过的两表联接没有什么本质区别。然而，有必要强调的是，被排序多次是因为每个联接条件都基于不同的列。举例来说，来自表t1和表t2之间的联接的数据结果，是根据表t1的id列来排序的，操作4会根据表t2的id列再次对其进行排序。同样的事情也发生在由操作3返回的数据上。事实上，它必须根据表t3的id列进行排序。总之，为处理这种类型的执行计划，必须执行六次排序，而且它们必须全部在能够返回任何一行数据之前执行。

14.3.4 工作区

为了处理合并联接，最多有两个内存中的工作区用于排序数据。如果一个排序被完全在内存中处理，就将其称为*内存中排序*（in-memory sort）。如果一个排序需要将临时数据溢出到磁盘上，就将其称为*磁盘上排序*（on-disk sort）。从性能的角度看，内存中排序显然比磁盘上排序要更快。下面会讨论这两种类型的排序如何工作。我还会讨论，如何根据dbms_xplan包的输出，识别哪一种排序用于处理一条SQL语句了。

我在第9章讨论过工作区配置（设置大小）。正如你可能从那一章回想起来的那样，有两种设置大小的方法。具体使用哪一种取决于workarea_size_policy初始化参数的取值。这两种方法如下所示。

- auto：数据库引擎自动调整工作区的大小。一个实例专有的PGA总量由pga_aggregate_target初始化参数控制，或者从11.1版本开始，由memory_target初始化参数控制。
- manual：sort_area_size初始化参数限制一个单独的工作区的大小。此外，sort_area_retained_size初始化参数控制当排序完成时如何释放PGA。

1. 内存中排序

内存中排序的处理过程直截了当。将数据加载到工作区后，排序就发生了。有一点值得强调的是，必须将所有数据都加载到工作区中，而不仅仅是作为联接条件引用的列。因此，为避免浪费大量的内存，SELECT子句中应该只引用真正有必要的列。为证实这一点，我们来看两个基于上一节中讨论的四表联接的例子。

在下面的例子中，所有表中的所有列都在SELECT子句中引用了。在执行计划中，OMem和Used-Mem列提供关于工作区的信息。前者是为内存中排序估算的内存总量。后者是在执行期间由操作使用的真实内存总量。括号中间的值（也就是0）意味着排序是完全在内存中进行的：

```
SELECT t1.*, t2.*, t3.*, t4.*
FROM t1, t2, t3, t4
WHERE t1.id = t2.t1_id
AND t2.id = t3.t2_id
AND t3.id = t4.t3_id
AND t1.n = 19
```

```
--------------------------------------------------------------
| Id  | Operation                | Name |  OMem | Used-Mem |
--------------------------------------------------------------
|   0 | SELECT STATEMENT         |      |       |          |
|   1 |  MERGE JOIN              |      |       |          |
|   2 |   SORT JOIN              |      | 34816 |30720  (0)|
|   3 |    MERGE JOIN            |      |       |          |
|   4 |     SORT JOIN            |      |  5120 | 4096  (0)|
|   5 |      MERGE JOIN          |      |       |          |
|   6 |       SORT JOIN          |      |  3072 | 2048  (0)|
|*  7 |        TABLE ACCESS FULL| T1   |       |          |
|*  8 |       SORT JOIN          |      | 21504 |18432  (0)|
|   9 |        TABLE ACCESS FULL| T2   |       |          |
|* 10 |     SORT JOIN            |      |   160K|  142K (0)|
|  11 |      TABLE ACCESS FULL   | T3   |       |          |
|* 12 |   SORT JOIN              |      |  1045K|  928K (0)|
|  13 |    TABLE ACCESS FULL     | T4   |       |          |
--------------------------------------------------------------

   7 - filter("T1"."N"=19)
   8 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")
  10 - access("T2"."ID"="T3"."T2_ID")
       filter("T2"."ID"="T3"."T2_ID")
  12 - access("T3"."ID"="T4"."T3_ID")
       filter("T3"."ID"="T4"."T3_ID")
```

在下面的例子中，SELECT子句中引用的列中只有一个没有被WHERE子句引用（t4.id）。有一点很重要，就是除了操作6，所有其他操作都使用了更小的工作区，而且这还是在两个案例中执行计划都是相同的情况下。还要注意查询优化器的估算（列Omem）考虑到了这个不同点：

```
SELECT t1.id, t2.id, t3.id, t4.id
FROM t1, t2, t3, t4
WHERE t1.id = t2.t1_id
AND t2.id = t3.t2_id
AND t3.id = t4.t3_id
AND t1.n = 19
```

```
--------------------------------------------------------------
| Id  | Operation                | Name |  OMem | Used-Mem |
--------------------------------------------------------------
|   0 | SELECT STATEMENT         |      |       |          |
|   1 |  MERGE JOIN              |      |       |          |
|   2 |   SORT JOIN              |      | 14336 |12288  (0)|
|   3 |    MERGE JOIN            |      |       |          |
|   4 |     SORT JOIN            |      |  3072 | 2048  (0)|
|   5 |      MERGE JOIN          |      |       |          |
|   6 |       SORT JOIN          |      |  3072 | 2048  (0)|
|*  7 |        TABLE ACCESS FULL| T1   |       |          |
|*  8 |       SORT JOIN          |      | 16384 |14336  (0)|
|   9 |        TABLE ACCESS FULL| T2   |       |          |
|* 10 |     SORT JOIN            |      |   106K|96256  (0)|
|  11 |      TABLE ACCESS FULL   | T3   |       |          |
```

```
|* 12 |    SORT JOIN            |      |   407K|  361K (0)|
|  13 |     TABLE ACCESS FULL   | T4   |       |          |
-----------------------------------------------------------

   7 - filter("T1"."N"=19)
   8 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")
  10 - access("T2"."ID"="T3"."T2_ID")
       filter("T2"."ID"="T3"."T2_ID")
  12 - access("T3"."ID"="T4"."T3_ID")
       filter("T3"."ID"="T4"."T3_ID")
```

2. 磁盘上排序

当一个工作区太小而不能包含所有数据时，数据库引擎会分多步处理排序。这些步骤在下面的列表中详述。显然，实际的步骤数量不仅依赖于数据总量，而且还依赖于工作区的大小。

(1) 从表中读取数据并存储到工作区中。当对其排序时，一个根据排序标准组织数据的结构被构建出来。在本例中，数据是根据id列进行排序的。这是图14-8中的第1步。

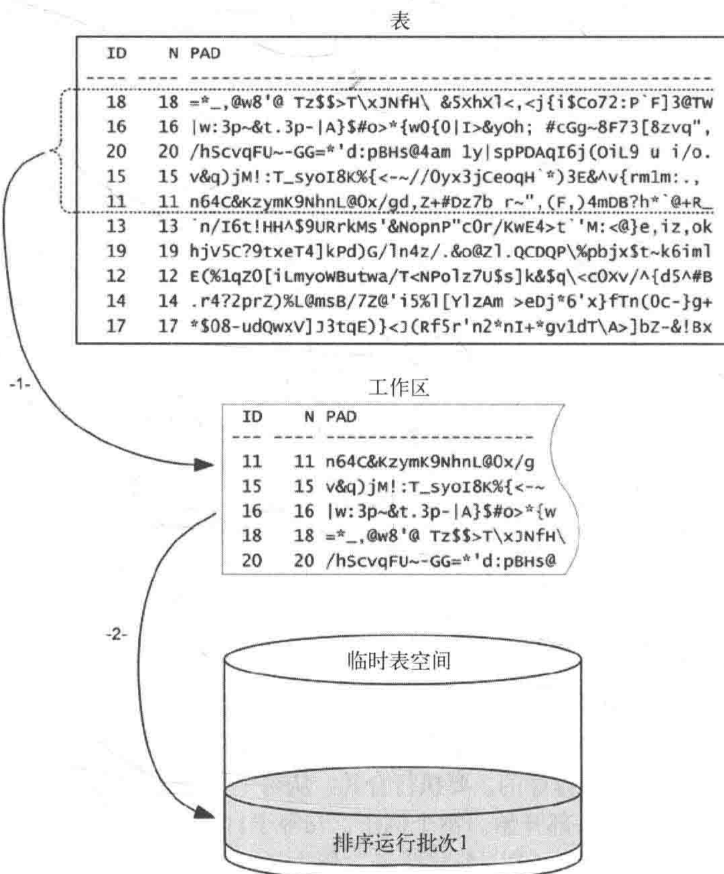


图14-8 磁盘上排序，排序运行批次1

(2) 填满工作区后，它的部分内容溢出到用户的临时表空间中的一个临时段中。这种类型的数据批次称作排序运行批次。注意，所有数据不仅存储在工作区中，而且还存储在临时段中。这是图14-8中的第2步。

(3) 既然数据已经溢出到临时段中，在工作区中就有了一些空闲空间。因此就可以继续在工作区中读取并排序输入的数据。这是图14-9中的第3步。

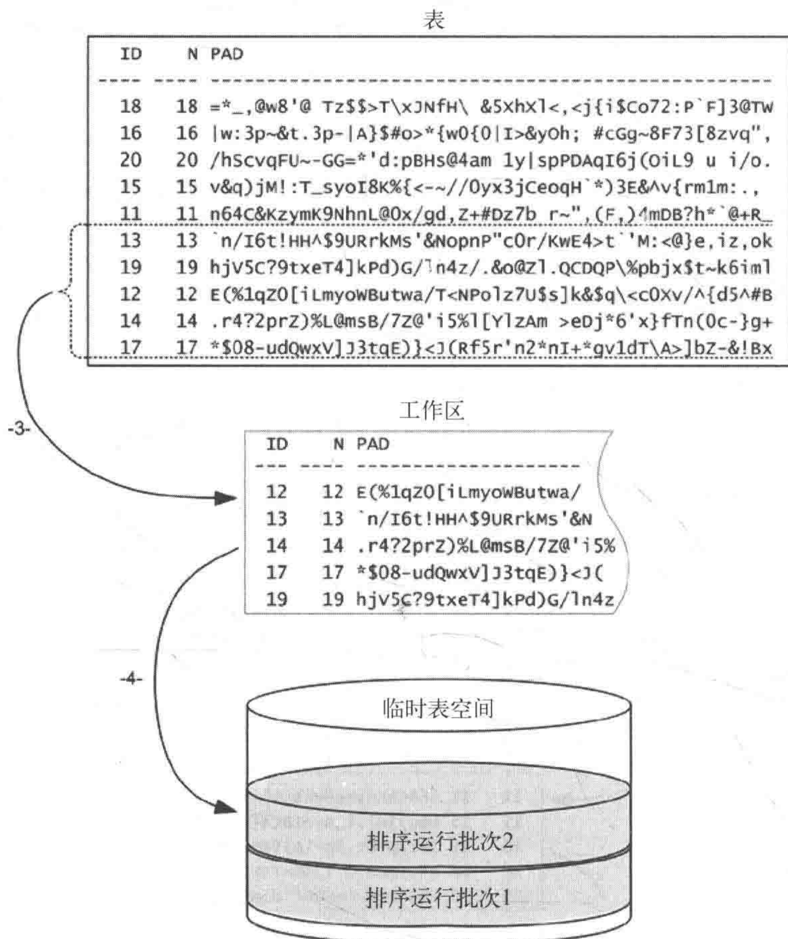


图14-9　磁盘上排序，排序运行批次2

(4) 再次填满工作区后，将另一个排序运行批次存储到临时段中。这是图14-9中的第4步。

(5) 对所有数据进行排序并存储到临时段中后，就该合并它了。合并阶段是必要的，因为每个排序运行批次都是独立于彼此进行排序的。要执行合并，从每个排序运行批次中读取回一些数据到工作区中，从每个排序运行批次的头部开始。举个例子，id等于11的记录存储在排序运行批次1中，而id等于12的记录存储在排序运行批次2中。换句话说，合并利用数据在溢出到临时段之前已被排序的事实，以便顺序读取每个排序运行批次。这是图14-10中的第5步。

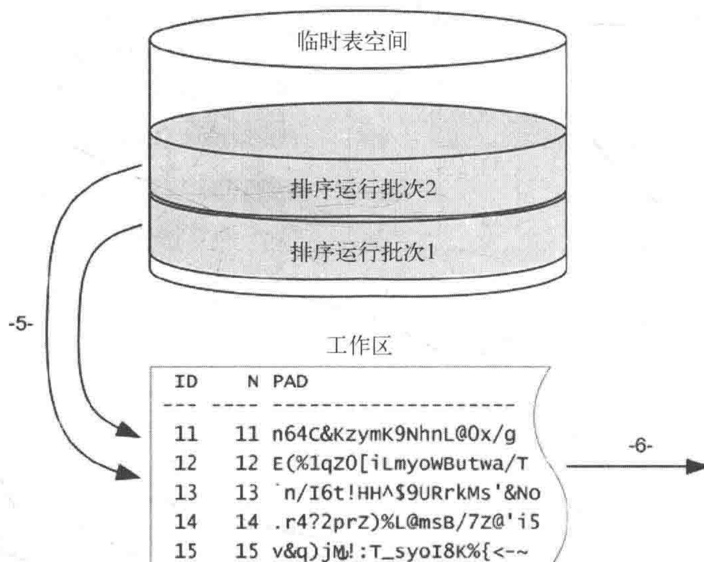


图14-10 磁盘上排序，合并阶段

(6) 一旦以正确方式排序的某些数据可以访问了，就能将它返回给父操作。这是图14-10中的第6步。

在刚刚描述的例子中，将数据写入到临时段中一次/从临时段中读取数据一次。这种类型的排序被称作一路排序（one-pass sort）。当工作区的大小远远小于需要排序的数据总量时，有必要经历多次合并阶段。在这种情况下，会多次将数据写入到临时段中/从临时段中多次读取数据。这种类型的排序称作多路排序（multipass sort）。显然，从性能的角度看，一路排序应该比多路排序更快。

要识别两种类型的排序，可以使用dbms_xplan包生成的输出。我们看一下来自已经在上一小节中使用的四表联接的输出。在这份输出中，显示了两个额外的列：1Mem和Used-Tmp。前者估算一路排序所需的内存总量。后者是在执行期间由操作使用的临时段的实际大小。如果没有值可用，那就意味着执行的是内存中排序。还有，注意对于使用临时空间的操作，括号之间的值不再是0。会将它们的值设置为执行排序的次数。换句话说，操作10是一路排序，而操作12是多路（9路）排序：

```
SELECT t1.*, t2.*, t3.*, t4.*
FROM t1, t2, t3, t4
WHERE t1.id = t2.t1_id
AND t2.id = t3.t2_id
AND t3.id = t4.t3_id
AND t1.n = 19

-----------------------------------------------------------------------------------------
| Id  | Operation              | Name | OMem  | 1Mem  | Used-Mem  | Used-Tmp|
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |      |       |       |           |         |
|   1 |  MERGE JOIN            |      |       |       |           |         |
|   2 |   SORT JOIN            |      | 34816 | 34816 |30720  (0)|    1024 |
|   3 |    MERGE JOIN          |      |       |       |           |         |
|   4 |     SORT JOIN          |      |  5120 |  5120 | 4096  (0)|         |
|   5 |      MERGE JOIN        |      |       |       |           |         |
```

```
|   6 |       SORT JOIN          |     | 3072 | 3072 | 2048  (0)|      |
|*  7 |        TABLE ACCESS FULL| T1  |      |      |          |      |
|*  8 |       SORT JOIN          |     | 9216 | 9216 |18432  (0)| 1024 |
|   9 |        TABLE ACCESS FULL| T2  |      |      |          |      |
|* 10 |     SORT JOIN            |     |  99K|  99K|32768  (1)| 1024 |
|  11 |      TABLE ACCESS FULL   | T3  |      |      |          |      |
|* 12 |   SORT JOIN              |     | 954K| 532K|41984  (9)| 2048 |
|  13 |    TABLE ACCESS FULL     | T4  |      |      |          |      |
-------------------------------------------------------------------------

   7 - filter("T1"."N"=19)
   8 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")
  10 - access("T2"."ID"="T3"."T2_ID")
       filter("T2"."ID"="T3"."T2_ID")
  12 - access("T3"."ID"="T4"."T3_ID")
       filter("T3"."ID"="T4"."T3_ID")
```

警告　通常dbms_xplan的输出以字节为单位显示关于内存大小的值。遗憾的是，就像在第10章中指出的那样，位于Used-Tmp列中的值必须乘以1024以转换为字节。举例来说，在上面的输出中，操作10和操作12分别使用了1 MB和2 MB的临时空间。

14.4　散列联接

本节介绍散列联接如何工作。首先会描述散列联接的常规行为并提供一些两表联接和四表联接的例子，接下来会对处理期间使用的工作区进行描述。最后，会介绍一种特殊的优化技术，索引联接。所有例子都基于hash_join.sql脚本。

14.4.1　概念

由散列联接处理的两组数据称作构建输入（build input）和探测输入（probe input）。构建输入是左输入，探测输入是右输入。如图14-11所示，使用构建输入的每一行，在内存中（如果没有足够的内存可用，也会使用临时表空间）构建出散列表。注意用于此用途的散列键是根据用作联接条件的列计算得来。一旦散列表包含来自构建输入的所有数据，就开始探测输入的处理过程。每一行都会针对散列表进行探测以找出它是否满足联接条件。显然，只有匹配的记录会被返回。

散列联接拥有的具体特征如下所示。

- 构建输入仅执行一次。
- 探测输入至多执行一次。如果构建输入不返回任何数据并且没有使用右外联接或全外联接，探测输入根本不会执行。
- 散列表仅构建于构建输入之上。所以，通常它构建于最小的输入上。
- 在返回第一行数据之前，只有构建输入需要被完全处理。
- 不支持交叉联接、内联接以及已分区外联接。

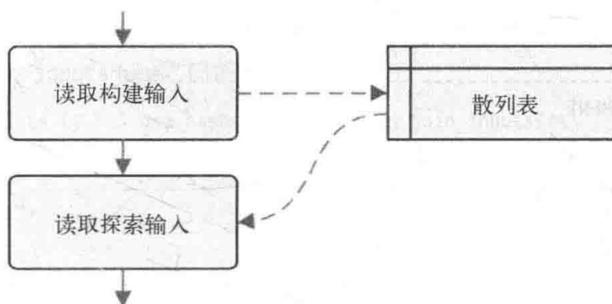


图14-11 由散列联接执行的处理过程概览

14.4.2 两表联接

下面是一个处理两表之间散列联接的简单执行计划。该例子还展示了如何依靠leading和use_hash hint来强制执行散列联接：

```
SELECT /*+ leading(t1 t2) use_hash(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 19

----------------------------------
| Id  | Operation          | Name |
----------------------------------
|   0 | SELECT STATEMENT   |      |
|*  1 |  HASH JOIN         |      |
|*  2 |   TABLE ACCESS FULL| T1   |
|   3 |   TABLE ACCESS FULL| T2   |
----------------------------------

   1 - access("T1"."ID"="T2"."T1_ID")
   2 - filter("T1"."N"=19)
```

如第10章中所描述的，HASH JOIN是无关联组合类型的操作。这意味着两个子操作至多被处理一次并且互相独立。在这种情况下，执行计划的处理过程可以总结为以下步骤。

- 表t1的所有数据通过一次全扫描读取，应用n=19这个限制条件，然后构建出一个使用该结果数据的散列表。为构建该散列表，会将一个散列函数应用到用作联接条件的列（id）上。
- 表t2的所有数据都通过一次全扫描读取，然后会将散列函数应用到用作联接条件的列上（t1_id），接下来探测该散列表。如果找到匹配，就返回结果数据。

HASH JOIN操作最重要的限制是其没有能力将索引应用于联接条件。这意味着索引仅能在指定限制条件的情况下用作访问路径。因此，为了选择访问路径，有必要为两张表都应用第10章中讨论的方法。举个例子，如果n=19这个限制条件提供强选择性，创建如下索引并应用可能会很有帮助：

```
CREATE INDEX t1_n ON t1 (n)
```

事实上，有了这个索引，就会使用下面的这个执行计划。注意，表t1不再通过全表扫描来访问了：

```
--------------------------------------------
| Id  | Operation                    | Name |
--------------------------------------------
|   0 | SELECT STATEMENT             |      |
|*  1 |  HASH JOIN                   |      |
|   2 |   TABLE ACCESS BY INDEX ROWID| T1   |
|*  3 |    INDEX RANGE SCAN          | T1_N |
|   4 |   TABLE ACCESS FULL          | T2   |
--------------------------------------------

  1 - access("T1"."ID"="T2"."T1_ID")
  3 - access("T1"."N"=19)
```

14.4.3 四表联接

下面的执行计划是一个典型的通过散列联接实现左深树（图示请参见图14-2）的例子。该例子还展示了如何使用leading和use_hash这两个hint来强制执行散列联接：

```
SELECT /*+ leading(t1 t2 t3 t4) use_hash(t2 t3 t4) */ t1.*, t2.*, t3.*, t4.*
FROM t1, t2, t3, t4
WHERE t1.id = t2.t1_id
AND t2.id = t3.t2_id
AND t3.id = t4.t3_id
AND t1.n = 19

------------------------------------
| Id  | Operation            | Name |
------------------------------------
|   0 | SELECT STATEMENT     |      |
|*  1 |  HASH JOIN           |      |
|*  2 |   HASH JOIN          |      |
|*  3 |    HASH JOIN         |      |
|*  4 |     TABLE ACCESS FULL| T1   |
|   5 |     TABLE ACCESS FULL| T2   |
|   6 |    TABLE ACCESS FULL | T3   |
|   7 |   TABLE ACCESS FULL  | T4   |
------------------------------------

  1 - access("T3"."ID"="T4"."T3_ID")
  2 - access("T2"."ID"="T3"."T2_ID")
  3 - access("T1"."ID"="T2"."T1_ID")
  4 - filter("T1"."N"=19)
```

这种类型的执行计划的处理过程如下所示。

- 表t1通过全扫描读取，应用n=19这个限制条件，然后创建包含结果数据的散列表。
- 表t2通过全扫描读取，探测上一步中创建的散列表。然后创建包含结果数据的散列表。
- 表t3通过全扫描读取，探测上一步中创建的散列表。然后创建包含结果数据的散列表。
- 表t4通过全扫描读取，探测上一步中创建的散列表。然后返回结果数据。仅当已全部处理完t1、t2和t3表时才可以返回第一行。反之，返回第一行数据之前没有必要全部处理完表t4。

散列联接的一个特别的属性是它还支持右深树和曲折树。下面的这个执行计划是前者的一个例子（图示请参见图14-3）。对比上一个例子，只有在SQL语句中指定的hint有所不同。注意，在这种情况下，

leading这个hint并不直接指定访问表的顺序（也就是t1 ➤ t2 ➤ t3 ➤ t4）。反之，它指定在应用要求交换左右输入的swap_join_inputs hint之前的顺序：

```
SELECT /*+ leading(t3 t4 t2 t1) use_hash(t1 t2 t4) swap_join_inputs(t1)
           swap_join_inputs(t2) */ t1.*, t2.*, t3.*, t4.*
FROM t1, t2, t3, t4
WHERE t1.id = t2.t1_id
AND t2.id = t3.t2_id
AND t3.id = t4.t3_id
AND t1.n = 19

-------------------------------------
| Id  | Operation            | Name |
-------------------------------------
|   0 | SELECT STATEMENT     |      |
|*  1 |  HASH JOIN           |      |
|*  2 |   TABLE ACCESS FULL  | T1   |
|*  3 |   HASH JOIN          |      |
|   4 |    TABLE ACCESS FULL | T2   |
|*  5 |    HASH JOIN         |      |
|   6 |     TABLE ACCESS FULL| T3   |
|   7 |     TABLE ACCESS FULL| T4   |
-------------------------------------

   1 - access("T1"."ID"="T2"."T1_ID")
   2 - filter("T1"."N"=19)
   3 - access("T2"."ID"="T3"."T2_ID")
   5 - access("T3"."ID"="T4"."T3_ID")
```

这两个执行计划（左深树和右深树）之间的区别之一是在给定的时间点上使用的活动工作区（散列表）的数量。使用左深树时，在同一时间最多有两工作区可使用。此外，当处理最后的表时，只需要一个单独工作区。另一方面，在右深树中，在几乎整个执行时间中都会分配和探测若干工作区（其数量等于联接的数量）。两个执行计划之间的另一个区别是它们的工作区大小。鉴于右深树工作区包含来自单张表的数据，而左深树工作区可以包含来自多张表联接的结果数据。因此，左深树工作区的大小依赖于联接是否限制返回的数据总量。

v$sql_workarea_active动态性能视图提供关于活动工作区的信息。下面的查询显示正在执行上面的执行计划的一个会话所使用的工作区。虽说operation_id列用于关联工作区和执行计划中的操作，actual_mem_used列显示其大小（按字节），而tempseg_size和tablespace 列则给出关于临时表空间使用情况的信息：

```
SQL> SELECT operation_id, operation_type, actual_mem_used, tempseg_size, tablespace
  2  FROM v$session s, v$sql_workarea_active w
  3  WHERE s.sid = w.sid
  4  AND s.sid = 24
  5  ORDER BY operation_id;

OPERATION_ID OPERATION_TYPE ACTUAL_MEM_USED TEMPSEG_SIZE TABLESPACE
------------ -------------- --------------- ------------ ----------
           1 HASH-JOIN                79872
           3 HASH-JOIN               161792
           5 HASH-JOIN               185344      1048576 TEMP
```

14

14.4.4　工作区

为了处理散列联接，会将内存中的工作区用于存储散列的数据。如果工作区足够大，能够存储整个散列表，则散列联接会完全在内存中处理。如果工作区不够大，则会将数据溢出到临时段中。（在本章前面部分解释过如何分辨完全在内存中执行的联接。）

我在第9章讨论过工作区配置（设置大小）。就像你可能回想起来的那样，有两种方法可用来执行设置大小。具体使用哪一种取决于workarea_size_policy初始化参数的值。

- auto：数据库引擎自动设置工作区的大小。一个实例拥有的PGA总量由pga_aggregate_target初始化参数控制，或者从11.1版本开始，由memory_target初始化参数控制。
- manual：hash_area_size初始化参数限制一个单独的工作区的最大大小。

14.4.5　索引联接

索引联接只能通过散列联接执行。因为这一点，可以认为它们是散列联接的特殊案例。其用途是通过联接属于相同表的两个或更多的索引来避免昂贵的表扫描。当一张表拥有许多被索引的列，并且其中几张表被一条SQL语句引用时，这个特性可能会非常有用。下面的查询是一个例子。注意这个查询如何引用一张单独的表，但是不管你期待的可能是什么，都会执行一个联接以取代单表访问。还有很重要的一点是，两个数据集之间的联接条件是基于rowid的。该例子还展示如何通过index_join这个hint 强制执行索引联接：

```
SELECT /*+ index_join(t4 t4_n t4_pk) */ id, n
FROM t4
WHERE id BETWEEN 10 AND 20
AND n < 100

----------------------------------------------
| Id  | Operation           | Name           |
----------------------------------------------
|   0 | SELECT STATEMENT    |                |
|*  1 |  VIEW               | index$_join$_001 |
|*  2 |   HASH JOIN         |                |
|*  3 |    INDEX RANGE SCAN| T4_N            |
|*  4 |    INDEX RANGE SCAN| T4_PK           |
----------------------------------------------

   1 - filter("ID"<=20 AND "N"<100 AND "ID">=10)
   2 - access(ROWID=ROWID)
   3 - access("N"<100)
   4 - access("ID">=10 AND "ID"<=20)
```

索引联接的一个有趣的限制条件是，如果在SELECT子句中引用了表的rowid，则查询优化器不会选择它们。因为rowid始终是索引的一部分，这个限制条件的起源是当前的实现，而不是因为索引本身缺乏必要的信息。

14.5 外联接

前面章节中描述的三种基本联接方法都支持外联接。当执行外联接时，在执行计划中唯一可见的区别就是追加到联接操作后面的OUTER关键字。为了演示，执行下面的SQL语句，因为hint的原因，执行计划使用了散列外联接。注意，即使SQL语句是使用新的联接语法书写的，谓词还是使用了基于(+)运算符的Oracle专有语法：

```
SELECT /*+ leading(t1) use_hash(t2) */ *
FROM t1 LEFT JOIN t2 ON t1.id = t2.t1_id

-----------------------------------
| Id  | Operation          | Name |
-----------------------------------
|   0 | SELECT STATEMENT   |      |
|*  1 |  HASH JOIN OUTER   |      |
|   2 |   TABLE ACCESS FULL| T1   |
|   3 |   TABLE ACCESS FULL| T2   |
-----------------------------------

   1 - access("T1"."ID"="T2"."T1_ID"(+))
```

除了是右外联接的散列联接，保留表（例如，上面SQL语句中的t1表）必须是联接操作的左输入。下面的执行计划，基于与上面例子相同的SQL语句，证实了这一点。在实践中，查询优化器选择在最小的结果集上构建散列表。当然了，这对于限制工作区的大小会有帮助。在本例中，因为表t1要小于表t2，出于演示的目的，有必要使用swap_join_inputs hint强制查询优化器交换两个联接输入：

```
SELECT /*+ leading(t1) use_hash(t2) swap_join_inputs(t2) */ *
FROM t1 LEFT JOIN t2 ON t1.id = t2.t1_id

--------------------------------------
| Id  | Operation             | Name |
--------------------------------------
|   0 | SELECT STATEMENT      |      |
|*  1 |  HASH JOIN RIGHT OUTER|      |
|   2 |   TABLE ACCESS FULL   | T2   |
|   3 |   TABLE ACCESS FULL   | T1   |
--------------------------------------

   1 - access("T1"."ID"="T2"."T1_ID"(+))
```

因此，无论何时查询优化器必须为一条SQL语句生成一个包含外联接的执行计划，除了散列联接，它的选择是非常有限的。

14.6 选择联接方法

要选择一种联接方法，必须考虑以下三个问题：

- ❑ 优化器目标，即first-rows优化和all-rows优化；
- ❑ 要优化的联接类型和谓词的选择率；

- 是否会以并行方式执行联接。

基于这三个准则，接下来的小节讨论如何选择一种联接方法，或者更具体点说，如何在嵌套循环联接、合并联接以及散列联接之间进行选择。

14.6.1 First-Rows 优化

使用first-rows优化时，总体响应时间是查询优化器的次要目标。返回开始若干行数据的响应时间显然是最重要的目标。因此，对于成功的first-rows优化，联接应该尽可能快地返回首先发现的匹配数据，而不是等待所有数据都处理完毕。出于这个目的，嵌套循环联接经常是最佳选择。散列联接，仅在某种程度上支持部分的联接，时而可用。比较起来，合并联接则很少适用于first-rows优化。

14.6.2 All-Rows 优化

使用all-rows优化，返回最后一行的响应时间是查询优化器最重要的目标。因此，对于成功的all-rows优化，联接应该尽可能迅速地完全执行。为选择最佳的联接方法，将逻辑读的数量减到最小，必须考虑是否有必要利用索引来应用联接条件。如果有必要通过一个索引来应用联接条件，就必须使用嵌套循环联接。否则，散列联接通常是最好的选择。通常来说，仅当结果集已排过序或因为技术限制（见14.6.3节）无法使用散列联接时，才会考虑合并联接。另一个可能比较有趣的合并联接的案例出现在可以避免排序操作的时候。当合并联接返回记录是按照`ORDER BY`子句指定的顺序时，就会发生这种情况。

14.6.3 支持的联接方法

要选择一种联接方法，必须要知道需要执行的联接的类型。事实上，并非所有联接方法都支持所有联接类型。表14-1总结了在不同情况下可用的联接方法。

表14-1　各联接方法支持的联接类型

联　接	嵌套循环联接	散列联接	合并联接
交叉联接	√		√
内联接	√		√
等值联接	√	√	√
半/反联接	√	√	√
外联接	√	√	√
已分区外联接	√		√

14.6.4 并行联接

所有联接方法都能以并行方式执行。不过，如图14-12所示，它们依不同比例决定。取决于并行度，一种方法可能会比另一种更快。所以，为选择出最佳的联接方法，一定要了解是否使用了并行处理以及并行度是多少。（第15章介绍并行处理。）

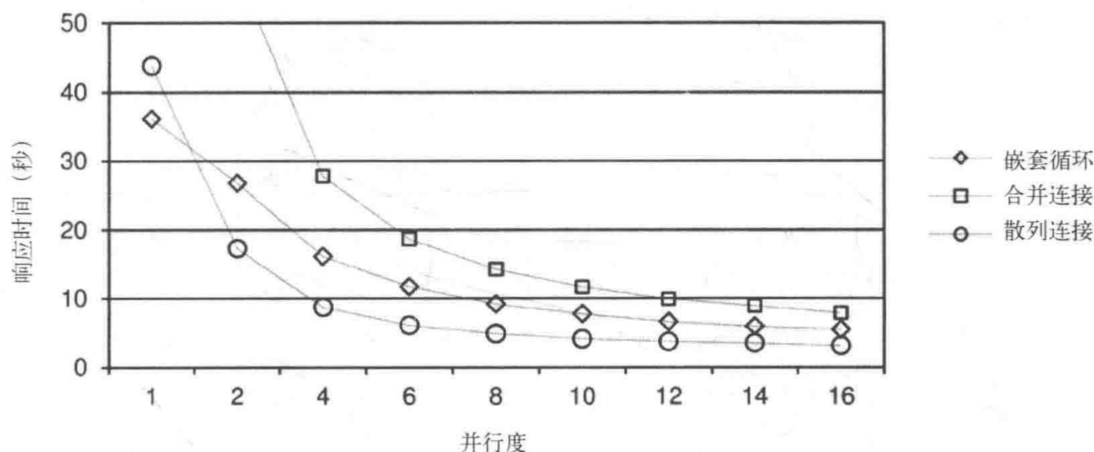


图14-12 不同并行度下的性能对比。此图展示的是在一个拥有8个内核的系统上执行的两表联接

14.7 分区智能联接

分区智能联接（partition-wise join，不要与已分区外联接混淆）是一项查询优化器仅会将其应用于合并和散列联接的优化技术。分区智能联接用于减少CPU、内存使用总量，以及，在RAC环境中，减少用于处理联接的网络资源。其基本思路是将一个大的联接拆分成多个较小的联接。分区智能联接可以是完全的或部分的。接下来的小节描述这两种选择。所有作为例子使用的查询都在脚本pwj.sql中提供。

注意 分区智能联接需要已分区表。因此，仅在使用了企业版中的分区选项时，这些技术才可用。

14.7.1 完全智能化分区联接

为了演示完全智能化分区联接的操作，我们首先来描述一下不使用这项优化技术的联接是怎么执行的。图14-13展示两张已分区表之间的一个联接。两张表所有数据行的一个单独的联接由一个单独的服务进程来执行。

当两张表在它们的联接键上是对等分区（equi-partitioned）时（例如，一张基于指向它的父表的外键参照分区的子表），数据库引擎能够利用完全智能化分区联接。取代执行一个单独的大联接，如图14-14所示，它执行多个较小的联接（在本例中，四个）。注意这样做可行是因为这两张表是以相同的方式分区的。因此，表1的分区1中存储的每一行仅在表2的分区1中有匹配的行。

在将一个大的联接分解为多个较小联接的过程中，其中一件最有用的事情就是提高执行过程并行化的可能性。事实上，数据库引擎能够为每个联接启动一个独立的从属进程。举例来说，在图14-14中服务进程协调四个从属进程来执行完全智能化分区联接。（第15章提供关于并行处理的详细信息。）

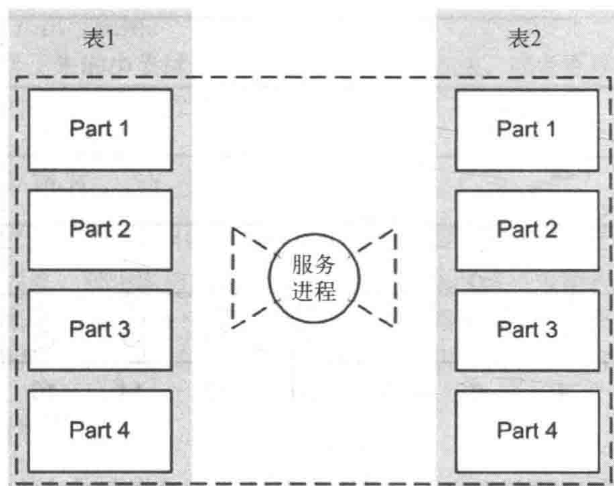


图14-13　不使用智能化分区联接来联接两张已分区表

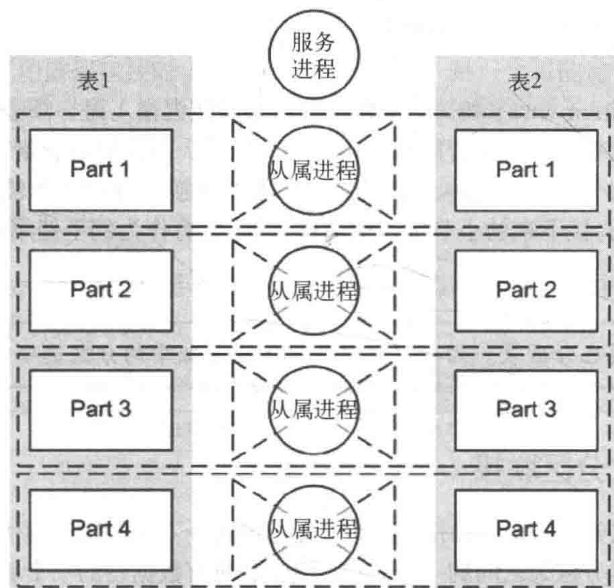


图14-14　使用智能化分区联接来联接两种已分区表

图14-15展示的是基于pwj_performance.sql脚本的一个性能测试的结果。这个测试的目的是重现类似图14-14中演示的那样的执行，或者，具体点说，重现拥有四个分区的两张表的一个联接。在这个特别的例子中，表中分别包含有10 000 000和100 000 000行数据。注意，并行执行的并行度等于分区的数量，即四个。

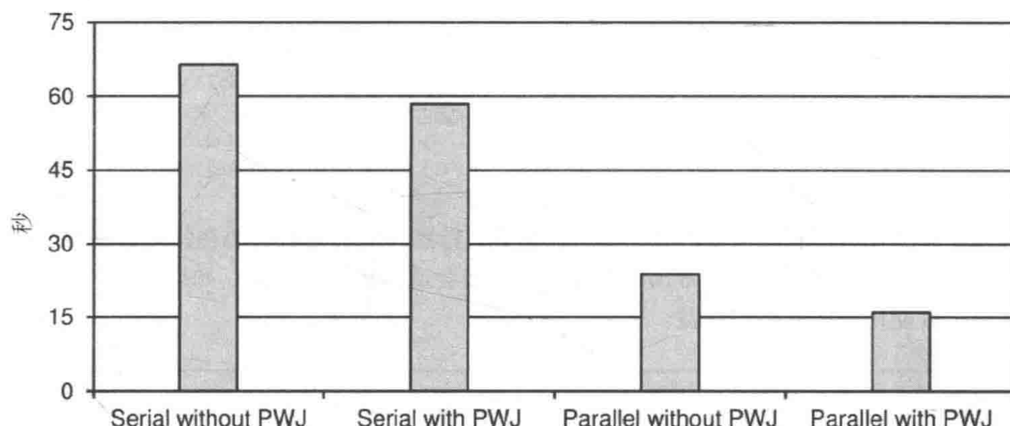


图14-15 两表联接在使用和不使用完全智能化分区联接的情况下分别的响应时间

为了识别是否使用了完全智能化分区联接，有必要查看一下执行计划。如果分区操作在联接操作之前出现，则意味着使用了完全智能化分区联接。在下面的执行计划中，PARTITION HASH ALL操作就是在HASH JOIN操作之前出现的：

```
SELECT *
FROM t1p, t2p
WHERE t1p.id = t2p.id

-----------------------------------
| Id  | Operation            | Name |
-----------------------------------
|   0 | SELECT STATEMENT     |      |
|   1 |  PARTITION HASH ALL  |      |
|*  2 |   HASH JOIN          |      |
|   3 |    TABLE ACCESS FULL| T1P  |
|   4 |    TABLE ACCESS FULL| T2P  |
-----------------------------------

   2 - access("T1P"."ID"="T2P"."ID")
```

上面的执行计划显示的是一个串行的完全智能化分区联接。接下来展示对于完全相同的SQL语句使用并行处理的执行计划。还有在这个案例中，PX PARTITION HASH ALL操作的出现要早于HASH JOIN操作。注意是如何使用pq_distribute hint来通知查询优化器使用完全智能化分区联接的（第15章提供关于用于并行处理的操作的详细信息）：

```
SELECT /*+ ordered use_hash(t2p) pq_distribute(t2p none none) */ *
FROM t1p, t2p
WHERE t1p.id = t2p.id

----------------------------------------
| Id  | Operation            | Name     |
----------------------------------------
|   0 | SELECT STATEMENT     |          |
|   1 |  PX COORDINATOR      |          |
|   2 |   PX SEND QC (RANDOM)| :TQ10000 |
```

```
|   3 |   PX PARTITION HASH ALL|          |
|*  4 |    HASH JOIN           |          |
|   5 |     TABLE ACCESS FULL  | T1P      |
|   6 |     TABLE ACCESS FULL  | T2P      |
-----------------------------------------

   4 - access("T1P"."ID"="T2P"."ID")
```

因为完全智能化分区联接需要两张对等分区的表，特别注意的是有必要在物理数据库设计期间就使用这项优化技术。换句话说，通常预期对等分区的表会遭遇大量的联接。如果不将它们对等分区，则无法从完全智能化分区联接中获益。

警告　当两张表在相同的值列表上定义了相同数量的分区时，且当分区按照相同的顺序进行定义时，会将两张列表已分区的表视为对等分区的。pwj_list.sql脚本演示了这个限制。

还有一件事情需要注意，所有的分区方法都是受支持的，并且完全智能化分区联接能够联接分区和子分区。举例而言，假如说你有两张表：sales和customers。联接键是customer_id。如果两张表都是散列分区的，拥有相同数量的分区，而且两张表都使用联接键作为分区键，则可以利用完全智能化分区联接。一定要记住，通常对一张类似sales这样的表（换句话说，一张包含历史数据的表）分区时需要的是范围分区。在这种情况下，可以为sales表使用组合分区。组合分区是通过在分区级别实施范围分区，在子分区级别使用散列分区来实现的，以满足两种需求。因此，就可以在customers表的散列分区和sales表的子分区之间执行完全智能化分区联接。

14.7.2　部分智能化分区联接

与完全智能化分区联接相比，部分智能化分区联接并不要求两张对等分区的表。此外，只有一张表必须是根据联接键进行分区的；另外一张表（可以分区也可以不分区）是根据联接键动态分区的。部分智能化分区联接的另外一个特性是它们只能够以并行方式执行。图14-16演示了一个部分智能化分区联接。在这个案例中，其中的一张表根本没有进行分区。在执行期间，数据库引擎启动两组并行从属进程。第一组读取非分区的表（表2）并根据分区键对数据进行分区。第二组接收来自第一组的数据，并且每一个从属进程读取已分区表的一个分区，然后执行它自己那部分联接。

要识别是否使用了部分智能化分区联接，有必要查看其执行计划。如果PX SEND操作是PARTITION (KEY)类型的，则意味着使用了部分智能化分区联接。在下面的例子中，操作7提供了这个信息：

```
SELECT /*+ ordered use_hash(t2p) pq_distribute(t2p none partition) */ *
FROM t1p, t2p
WHERE t1p.id = t2p.id

---------------------------------------------
| Id  | Operation                  | Name     |
---------------------------------------------
|   0 | SELECT STATEMENT           |          |
|   1 |  PX COORDINATOR            |          |
|   2 |   PX SEND QC (RANDOM)      | :TQ10001 |
|*  3 |    HASH JOIN BUFFERED      |          |
```

```
|  4 |     PX PARTITION HASH ALL    |         |
|  5 |      TABLE ACCESS FULL       | T1P     |
|  6 |     PX RECEIVE               |         |
|  7 |      PX SEND PARTITION (KEY)| :TQ10000 |
|  8 |       PX BLOCK ITERATOR      |         |
|  9 |        TABLE ACCESS FULL     | T2P     |
-----------------------------------------------

   3 - access("T1P"."ID"="T2P"."ID")
```

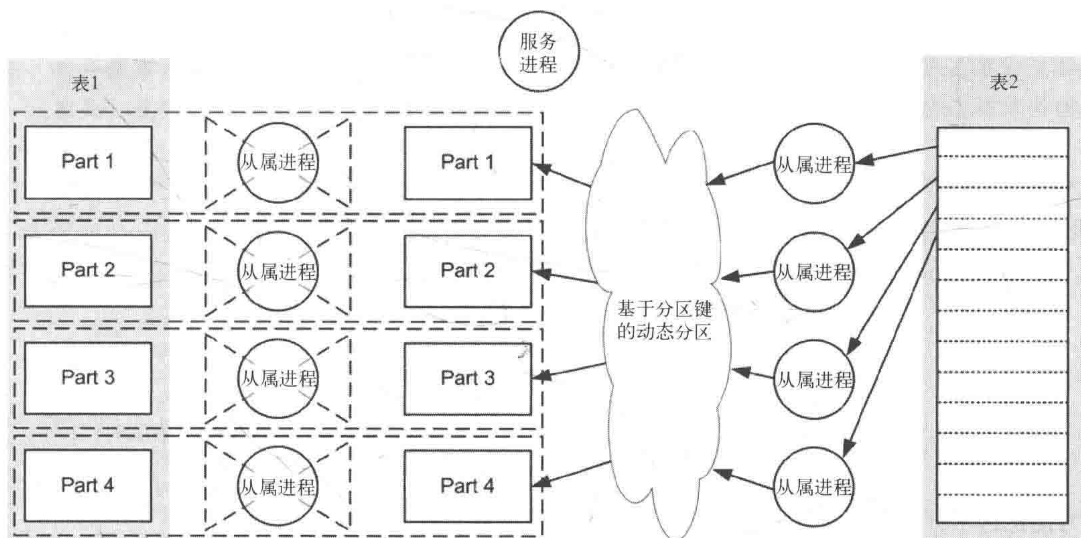


图14-16　使用部分智能化分区联接来联接两张表

在实践中，部分智能化分区联接并不一定会引起性能的提升。事实上，正常的联接可能会比部分智能化分区联接更快。因为使用部分智能化分区联接可能会对性能不利，一般很少会看见查询优化器使用这种优化技术。

14.8　星型转换

星型转换（star transformation）是一项用于星型模式（star schema，也称作维度模型）的优化技术。这种类型的模式是由一个大的中央表（事实表）以及几个其他的表（维度表）组成的。它的主要特点是事实表依赖维度表。图14-17是一个基于样例模式SH（*Sample Schemas*手册对此有完整描述）的例子。

注意　星型转换仅在企业版中可用。要想在标准版中利用一种类似的优化技术，必须自己手动重写查询。在本节的结尾会提供这样的一个重写的例子。

下面是一个典型的针对星型模式执行的查询（注意，没有将限制条件应用于事实表上，而只是应用于维度表上）：

```
SELECT c.cust_state_province, t.fiscal_month_name, sum(s.amount_sold) AS amount_sold
FROM sales s, customers c, times t, products p
WHERE s.cust_id = c.cust_id
AND s.time_id = t.time_id
AND s.prod_id = p.prod_id
AND c.cust_year_of_birth BETWEEN 1970 AND 1979
AND p.prod_subcategory = 'Cameras'
GROUP BY c.cust_state_province, t.fiscal_month_name
ORDER BY c.cust_state_province, sum(s.amount_sold) DESC
```

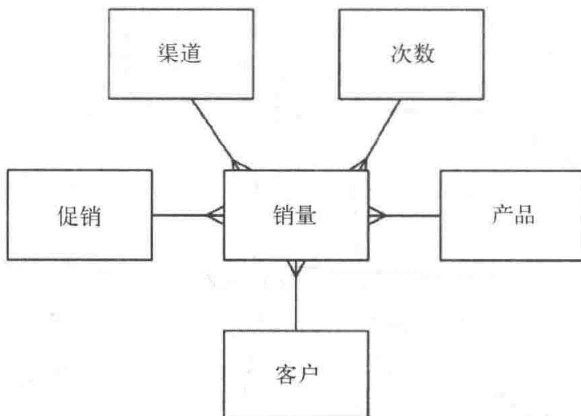


图14-17　一个典型的星型模式

为优化这个针对星型模式的查询，查询优化器应该完成以下几件事。

(1) 开始评估在其上面含有限制条件的每张维度表。

(2) 使用产生的维度键装配一个列表。

(3) 使用该列表从事实表中提取匹配的记录。

遗憾的是，这种方法无法使用正常的联接实现。一方面，查询优化器在同一时间只能联接两组数据。另一方面，联接两张维度表会导致笛卡儿积的出现。为了解决这个问题，Oracle数据库实现了星型转换。

警告　尽管星型转换是在1997年的8.0版本中首度引入的，并且在两年后的8.1版本中得到极大的增强，但是，在之前的很长一段时间，它的稳定性一直是一个问题。可能自从其引入之后发布的每个补丁集都会修复一些与其相关的bug。一旦有什么不对的地方，就会生成类似ORA-07445、ORA-00600、ORA-00942的错误，或出现不正确的结果。尽管如此，自从8.1.6版本开始就可以顺利地使用这个特性了。Oracle的查询优化器开发组很清楚这个问题，并为11.2版本完全重写了这部分代码。因此，在最近的版本中它的稳定性得到了改善。我的建议很简单，就是仔细对其进行测试。如果它运行没问题，可以考虑其在性能方面带来的提高。如果有问题，至少你在将其用于生产环境之前就会知道。我还建议你查看Oracle Support文档47358.1（*Init.ora Parameter STAR_TRANSFORMATION_ENABLED Reference Note*）。此文档给出影响每个具体版本的一个bug列表。

你需要满足两个基本的要求才能从星型转换中获益。首先，必须启用该特性。可以使用star_transformation_enabled初始化参数来控制它。注意，因为默认情况下会将这个参数设置为FALSE，所以默认会禁用该特性。要启用它，应该将它设置为temp_disable或TRUE。其次，在事实表上，引用维度表的每个联接条件上必须都建一个索引。联接条件不必基于外键，但是如果外键确实存在，它们有助于查询优化器找到一个最优的执行计划。尽管查询优化器可以在运行时将B树索引转化成位图索引，执行计划使用位图索引会更加高效。总之，我建议出于最佳性能的考虑，在每一个联接条件上创建外键和位图索引。

提示 当一张事实表和一张维度表之间的联接条件是基于多个列时，星型转换有时是不受支持的。因此，我强烈推荐你在一个单独的列上使用联接条件，并进而也基于单独的列创建外键和位图索引。

将star_transformation_enabled初始化参数设置为temp_disable时，就会为之前展示的SQL语句使用以下执行计划。这个例子，与接下来的例子一样，都来自star_transformation.sql脚本：

```
---------------------------------------------------------------------------
| Id  | Operation                              | Name                     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT                       |                          |
|   1 |  SORT ORDER BY                         |                          |
|   2 |   HASH GROUP BY                        |                          |
|*  3 |    HASH JOIN                           |                          |
|   4 |     TABLE ACCESS FULL                  | TIMES                    |
|*  5 |     HASH JOIN                          |                          |
|*  6 |      TABLE ACCESS FULL                 | CUSTOMERS                |
|   7 |      VIEW                              | VW_ST_FE4FBDB9           |
|   8 |       NESTED LOOPS                     |                          |
|   9 |        BITMAP CONVERSION TO ROWIDS     |                          |
|  10 |         BITMAP AND                     |                          |
|  11 |          BITMAP MERGE                  |                          |
|  12 |           BITMAP KEY ITERATION         |                          |
|  13 |            TABLE ACCESS BY INDEX ROWID| PRODUCTS                 |
|* 14 |             INDEX RANGE SCAN           | PRODUCTS_PROD_SUBCAT_IX  |
|* 15 |            BITMAP INDEX RANGE SCAN     | SALES_PROD_BIX           |
|  16 |          BITMAP MERGE                  |                          |
|  17 |           BITMAP KEY ITERATION         |                          |
|* 18 |            TABLE ACCESS FULL           | CUSTOMERS                |
|* 19 |            BITMAP INDEX RANGE SCAN     | SALES_CUST_BIX           |
|  20 |        TABLE ACCESS BY USER ROWID      | SALES                    |
---------------------------------------------------------------------------

   3 - access("ITEM_1"="T"."TIME_ID")
   5 - access("ITEM_2"="C"."CUST_ID")
   6 - filter("C"."CUST_YEAR_OF_BIRTH">=1970 AND "C"."CUST_YEAR_OF_BIRTH"<=1979)
  14 - access("P"."PROD_SUBCATEGORY"='Cameras')
  15 - access("S"."PROD_ID"="P"."PROD_ID")
  18 - filter("C"."CUST_YEAR_OF_BIRTH">=1970 AND "C"."CUST_YEAR_OF_BIRTH"<=1979)
  19 - access("S"."CUST_ID"="C"."CUST_ID")
```

因为这个执行计划包含一些特别的操作，我们来仔细看一下它的操作。

(1) 执行始于操作4，对times维度表的全表扫描。通过由它返回的数据，一个散列表被操作3执行的散列联接建立起来。

(2) 操作6对customers维度表做了一次全表扫描并应用c.cust_year_of_birth BETWEEN 1970 AND 1979这个限制条件。通过由它返回的数据，一个散列表被操作5执行的散列联接建立起来。

(3) 操作13和14访问products维度表并应用p.prod_subcategory='Cameras'这个限制条件。

(4) 操作12 BITMAP KEY ITERATION是一个关联组合操作。对于由其第一个子操作（操作13）返回的数据，第二个子操作（操作15）就被执行一次。在此情况下，就对基于在事实表上定义的位图索引执行了一次检索。

(5) 操作11 BITMAP MERGE合并由它的子操作传递过来的位图。这个操作是必要的，因为来自于一个位图索引的索引键可能只会覆盖被索引表的一部分。

(6) 操作16到19按照与操作11到15处理products维度表相同的方式处理customers维度表。事实上，每个应用了限制条件的维度都是按照相同的方式处理的。

(7) 操作10 BITMAP AND组合从它的两个子操作（11和16）传递过来的位图并且仅保留匹配的条目。

(8) 操作9 BITMAP CONVERSION TO ROWIDS将从它的子操作（10）传递过来的位图转换为sales事实表的rowid。

(9) 操作20因嵌套循环联接（操作8）而访问包含由操作9生成的rowid的事实表。

(10) 操作7仅是提供信息的，告诉你查询块包含一个来自星型转换的不可合并的结果视图（注意视图名称的VW_ST前缀）。这个操作仅从11.2.0.2版本开始才可用。

(11) 通过由操作8返回的记录，对两个散列联接（操作3和操作5）的散列表进行探测。如果找到匹配的记录，会将它们传递给操作2。

(12) 操作2 HASH GROUP BY处理GROUP BY子句并将结果数据发送给操作1。

(13) 最后，操作1 SORT ORDER BY处理ORDER BY子句。

概括起来，完成一次星型转换会执行以下步骤。

(1) 会将维度表与事实表对应的位图索引相“联接”。这个操作仅当维度表拥有应用于它们的限制条件时才是必要的，在本例中，是products和customers表。

(2) 会将结果位图合并转化为rowid。然后通过这些rowid访问事实表。

(3) 会将维度表与从事实表中提取出来的数据进行联接。这个操作仅当维度表在WHERE子句之外拥有被引用的列时才是必要的，在本例中，是times和customers表。这就是customers表会在执行计划中出现两次的原因。

你可以为这个基本的行为应用另外两种额外的优化技术：临时表和位图联接索引。

临时表的目的是避免对维度表的双重处理。例如，在上面的执行计划中，不仅customers维度表被通过全表扫描访问了两次（操作6和操作18），而且应用于它的谓词也被执行了两次。思路是只访问每个维度一次，应用谓词，并将结果数据存储在一张临时表中。这项优化技术会在star_transformation_enabled初始化参数设置为TRUE时启用。下面的执行计划是一个例子，也是基于与之前相同的SQL语句。注意sys_temp_0fd9d6647_1cb85c临时表的创建（操作1到操作3）和它的使用（操作7和操作21）：

```
-----------------------------------------------------------------------------
| Id  | Operation                               | Name                      |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT                        |                           |
|   1 |  TEMP TABLE TRANSFORMATION              |                           |
|   2 |   LOAD AS SELECT                        | SYS_TEMP_0FD9D6647_1CB85C |
|*  3 |    TABLE ACCESS FULL                    | CUSTOMERS                 |
|   4 |   SORT ORDER BY                         |                           |
|   5 |    HASH GROUP BY                        |                           |
|*  6 |     HASH JOIN                           |                           |
|   7 |      TABLE ACCESS FULL                  | SYS_TEMP_0FD9D6647_1CB85C |
|*  8 |      HASH JOIN                          |                           |
|   9 |       TABLE ACCESS FULL                 | TIMES                     |
|  10 |       VIEW                              | VW_ST_16AF99B7            |
|  11 |        NESTED LOOPS                     |                           |
|  12 |         BITMAP CONVERSION TO ROWIDS     |                           |
|  13 |          BITMAP AND                     |                           |
|  14 |           BITMAP MERGE                  |                           |
|  15 |            BITMAP KEY ITERATION         |                           |
|  16 |             TABLE ACCESS BY INDEX ROWID| PRODUCTS                  |
|* 17 |              INDEX RANGE SCAN           | PRODUCTS_PROD_SUBCAT_IX   |
|* 18 |             BITMAP INDEX RANGE SCAN     | SALES_PROD_BIX            |
|  19 |           BITMAP MERGE                  |                           |
|  20 |            BITMAP KEY ITERATION         |                           |
|  21 |             TABLE ACCESS FULL           | SYS_TEMP_0FD9D6647_1CB85C |
|* 22 |             BITMAP INDEX RANGE SCAN     | SALES_CUST_BIX            |
|  23 |         TABLE ACCESS BY USER ROWID      | SALES                     |
-----------------------------------------------------------------------------

   3 - filter("C"."CUST_YEAR_OF_BIRTH">=1970 AND "C"."CUST_YEAR_OF_BIRTH"<=1979)
   6 - access("ITEM_2"="C0")
   8 - access("ITEM_1"="T"."TIME_ID")
  17 - access("P"."PROD_SUBCATEGORY"='Cameras')
  18 - access("S"."PROD_ID"="P"."PROD_ID")
  22 - access("S"."CUST_ID"="C0")
```

这第二项优化技术基于位图联接索引。思路是避免维度表与对应的事实表上的位图索引之间的"联接"。出于这个目的，位图联接索引必须在事实表上进行创建，并对维度表的一个或多个列进行索引。例如，下面的索引分别对于应用c.cust_year_of_birth BETWEEN 1970 AND 1979和p.prod_ subcategory = 'Cameras'限制条件是有必要的：

```
CREATE BITMAP INDEX sales_cust_year_of_birth_bix ON sales (c.cust_year_of_birth)
FROM sales s, customers c
WHERE s.cust_id = c.cust_id

CREATE BITMAP INDEX sales_prod_subcategory_bix ON sales (p.prod_subcategory)
FROM sales s, products p
WHERE s.prod_id = p.prod_id
```

创建了这两个索引之后，就产生了以下执行计划。注意，用于产生rowid的方法（第8~12行）要比在上一个例子中使用的那个直截了当得多。实际上，取代访问维度表，并将它们与事实表上的位图索引进行联接，仅访问位图联接索引就足够了。这样做可行的原因是与维度数据有关的值已经呈现在

事实表的位图联接索引中了：

```
----------------------------------------------------------------------------
| Id  | Operation                            | Name                          |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT                     |                               |
|   1 |  SORT ORDER BY                       |                               |
|   2 |   HASH GROUP BY                      |                               |
|*  3 |    HASH JOIN                         |                               |
|   4 |     TABLE ACCESS FULL                | TIMES                         |
|*  5 |     HASH JOIN                        |                               |
|*  6 |      TABLE ACCESS FULL               | CUSTOMERS                     |
|   7 |      TABLE ACCESS BY INDEX ROWID     | SALES                         |
|   8 |       BITMAP CONVERSION TO ROWIDS    |                               |
|   9 |        BITMAP AND                    |                               |
|* 10 |         BITMAP INDEX SINGLE VALUE    | SALES_PROD_SUBCATEGORY_BIX    |
|  11 |         BITMAP MERGE                 |                               |
|* 12 |          BITMAP INDEX RANGE SCAN     | SALES_CUST_YEAR_OF_BIRTH_BIX  |
----------------------------------------------------------------------------

   3 - access("S"."TIME_ID"="T"."TIME_ID")
   5 - access("S"."CUST_ID"="C"."CUST_ID")
   6 - filter("C"."CUST_YEAR_OF_BIRTH">=1970 AND "C"."CUST_YEAR_OF_BIRTH"<=1979)
  10 - access("S"."SYS_NC00009$"='Cameras')
  12 - access("S"."SYS_NC00008$">=1970 AND "S"."SYS_NC00008$"<=1979)
```

星型转换是一种基于成本的查询转换。因此，当启用时，查询优化器不仅决定使用星型转换是否合理，而且决定临时表和/或位图联接索引是否有助于SQL语句的高效执行。这个特性的使用也可以通过hint star_transformation和no_star_transformation控制。

如果正在使用标准版，那么星型转换和位图索引都是不可用的。在这种情况下，想要获得较好的性能，可能需要自己手动改写查询。如下面的例子所示，尽管重写的查询可读性较差，执行计划却十分相似：

```
SELECT c.cust_state_province, t.fiscal_month_name, sum(s.amount_sold) AS amount_sold
FROM (SELECT *
      FROM sales
      WHERE rowid IN (SELECT c.rid
                      FROM (SELECT s.rowid AS rid
                            FROM customers c, sales s
                            WHERE c.cust_id = s.cust_id
                            AND c.cust_year_of_birth BETWEEN 1970 AND 1979) c,
                            (SELECT s.rowid AS rid
                            FROM products p, sales s
                            WHERE p.prod_id = s.prod_id
                            AND p.prod_subcategory = 'Cameras') p
                      WHERE c.rid = p.rid)) s,
     customers c, times t

WHERE s.cust_id = c.cust_id
AND s.time_id = t.time_id
GROUP BY c.cust_state_province, t.fiscal_month_name
ORDER BY c.cust_state_province, sum(s.amount_sold) DESC
```

```
-------------------------------------------------------------------------------
| Id  | Operation                                     | Name                   |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                              |                        |
|   1 |  SORT ORDER BY                                |                        |
|   2 |   HASH GROUP BY                               |                        |
|*  3 |    HASH JOIN                                  |                        |
|   4 |     TABLE ACCESS FULL                         | TIMES                  |
|*  5 |     HASH JOIN                                 |                        |
|   6 |      TABLE ACCESS FULL                        | CUSTOMERS              |
|   7 |      VIEW                                     |                        |
|   8 |       NESTED LOOPS                            |                        |
|   9 |        VIEW                                   | VW_NSO_1               |
|  10 |         HASH UNIQUE                           |                        |
|* 11 |          HASH JOIN                            |                        |
|  12 |           VIEW                                |                        |
|  13 |            NESTED LOOPS                       |                        |
|  14 |             TABLE ACCESS BY INDEX ROWID       | PRODUCTS               |
|* 15 |              INDEX RANGE SCAN                 | PRODUCTS_PROD_SUBCAT_IX |
|* 16 |             INDEX RANGE SCAN                  | SALES_PROD_BIX         |
|  17 |           VIEW                                |                        |
|  18 |            NESTED LOOPS                       |                        |
|* 19 |             TABLE ACCESS FULL                 | CUSTOMERS              |
|* 20 |             INDEX RANGE SCAN                  | SALES_CUST_BIX         |
|  21 |        TABLE ACCESS BY USER ROWID             | SALES                  |
-------------------------------------------------------------------------------

   3 - access("S"."TIME_ID"="T"."TIME_ID")
   5 - access("S"."CUST_ID"="C"."CUST_ID")
  11 - access("C"."RID"="P"."RID")
  15 - access("P"."PROD_SUBCATEGORY"='Cameras')
  16 - access("P"."PROD_ID"="S"."PROD_ID")
  19 - filter("C"."CUST_YEAR_OF_BIRTH">=1970 AND
              "C"."CUST_YEAR_OF_BIRTH"<=1979)
  20 - access("C"."CUST_ID"="S"."CUST_ID")
```

14.9 小结

本章描述与联接有关的两个主题。第一，本章涵盖了数据库引擎执行联接（嵌套循环联接、合并联接和散列联接）时所使用的方法，并讨论了什么时候使用它们是合理的。第二，本章涵盖了一些适用于查询优化器改进性能的优化技术。

现在我已经讨论了基础的访问路径和联接方法，是时候关注高级优化技术了。在下一章中，我会讨论物化视图、结果缓存、并行处理以及直接路径插入。所有的这些特性都不会经常用到，但是正确使用它们时，能够带来极大的性能改进。是时候超越对访问和联接的优化了。

第 15 章 数据访问和联接优化之外

在考虑本章呈现的高级优化技术之前，必须首先完成对数据访问和联接的优化。实际上，只有通过其他方式无法进行优化的时候，才会考虑使用此处描述的优化技术进一步改进性能。换句话说，应该首先修复基础问题，然后，如果性能表现仍然不可接受，可以考虑特别的手段。

本章将会讲述物化视图、结果缓存、并行处理、直接路径插入、行预取和数据接口等技术如何运作，以及可以如何将它们用于改进性能。描述其中每一种优化技术的章节都会按照相同的方式进行组织。一个简短的介绍，紧接着描述这种技术是如何运作的，然后是应该在什么时候使用它。所有小节都以对一些常见的陷阱和谬误的讨论作为结尾。

注意 本章会有一些包含hint的SQL语句作为例子向你展示它们的使用方法。不管怎样，这里并没有提供真实的参考和完整的语法。可以在*Oracle Database SQL Language Reference*手册的第2章中找到这些说明。

本章会展示各种不同的性能测试的结果。这些性能数据只是试图帮助你对比不同类型的处理方式，并且提供关于它们的影响的直观感受。记住，每个系统和每个应用程序都拥有自己的特性。因此，使用每种技术的相关性可能会有很大不同，具体取决于在哪里应用这种技术。

15.1 物化视图

视图是一张根据在创建视图时指定的查询语句返回结果集的虚表。每次访问视图时，就会执行查询。为了避免在每次访问时都执行查询，可以将查询的结果集存储在一张物化视图中。换句话说，物化视图仅仅转换并复制本来已经存储在别处的数据。

注意 物化视图也可用于分布式环境中，以便在数据库之间复制数据。这种用法本书不作介绍。

15.1.1 工作原理

接下来的小节会描述物化视图是什么以及它是如何运作的。在描述完物化视图的基础概念之后，会详细论述查询重写和刷新。

1. 概念

假设必须为查询（在mv.sql脚本中提供）改进性能，该查询基于简单的模式对象sh（*Oracle Database Sample Schemas*手册完整描述了此模式对象）：

```
SELECT p.prod_category, c.country_id,
       sum(quantity_sold) AS quantity_sold,
       sum(amount_sold) AS amount_sold
FROM sales s, customers c, products p
WHERE s.cust_id = c.cust_id
AND s.prod_id = p.prod_id
GROUP BY p.prod_category, c.country_id
ORDER BY p.prod_category, c.country_id
```

如果按照第10章和第13章中描述的方法和规则评估该执行计划的效率，你会发现一切正常。估算很完美，不同访问路径每返回一行的逻辑读也很低：

```
-----------------------------------------------------------------------------------
| Id  | Operation                | Name      | Starts | E-Rows | A-Rows | Buffers |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |           |      1 |        |     81 |    3094 |
|   1 |  SORT GROUP BY           |           |      1 |     68 |     81 |    3094 |
|*  2 |   HASH JOIN              |           |      1 |    968 |    956 |    3094 |
|   3 |    TABLE ACCESS FULL     | PRODUCTS  |      1 |     72 |     72 |       3 |
|   4 |    VIEW                  | VW_GBC_9  |      1 |    968 |    956 |    3091 |
|   5 |     HASH GROUP BY        |           |      1 |    968 |    956 |    3091 |
|*  6 |      HASH JOIN           |           |      1 |    918K|    918K|    3091 |
|   7 |       TABLE ACCESS FULL  | CUSTOMERS |      1 |  55500 |  55500 |    1456 |
|   8 |       PARTITION RANGE ALL|           |      1 |    918K|    918K|    1635 |
|   9 |        TABLE ACCESS FULL | SALES     |     28 |    918K|    918K|    1635 |
-----------------------------------------------------------------------------------

   2 - access("ITEM_1"="P"."PROD_ID")
   6 - access("S"."CUST_ID"="C"."CUST_ID")
```

“问题”是在聚合发生之前处理了大量的数据。无法只通过改变某条访问路径或某个联接方法就能使性能有所增强，因为它们已经被最大程度地优化了；换句话说，它们的全部潜力已经被开发完毕。是时候应用一种高级优化技术了。我们来根据要优化的查询创建一张物化视图。

物化视图是通过CREATE MATERIALIZED VIEW语句创建的。在最简单的情况下，需要指定一个名称和构建物化视图所依据的查询。注意物化视图依据的表叫作基表（也称为主表）。下面的SQL语句和图15-1对此进行了演示（注意，在原始查询中的ORDER BY子句被省略了）：

```
CREATE MATERIALIZED VIEW sales_mv
AS
SELECT p.prod_category, c.country_id,
       sum(quantity_sold) AS quantity_sold,
       sum(amount_sold) AS amount_sold
FROM sales s, customers c, products p
WHERE s.cust_id = c.cust_id
AND s.prod_id = p.prod_id
GROUP BY p.prod_category, c.country_id
```

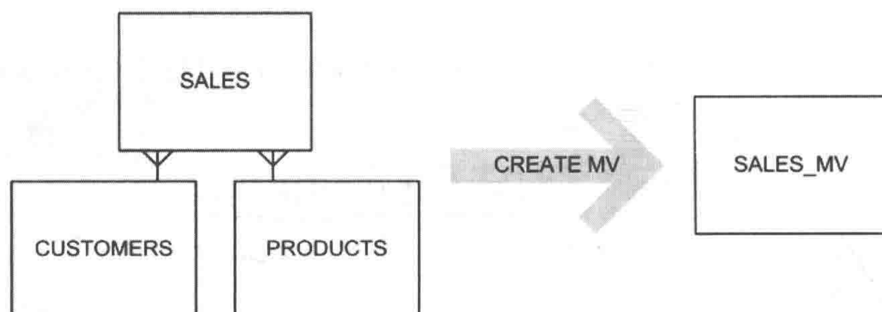


图15-1　物化视图的创建

注意　当你基于包含ORDER BY子句的查询创建一张物化视图时，只有在物化视图创建的时候会根据ORDER BY子句对数据进行排序。随后，在刷新期间，不会一直保持这个排序准则。这是因为物化视图定义之中不会包含ORDER BY子句，也不会将其存储到数据字典中。

当你执行上面的SQL语句时，数据库引擎就会创建一张物化视图（其只是数据字典中的一个对象，换句话说，它只是元数据）和一张容器表。该容器表是一张拥有与物化视图相同名称的普通堆表。容器表用于存储由查询返回的结果集。

可以像查询其他任何表那样查询容器表。下面的SQL语句展示这样的一个例子：

```
SELECT *
FROM sales_mv
ORDER BY prod_category, country_id

------------------------------------------------------------------------------------
| Id  | Operation              | Name     | Starts | E-Rows | A-Rows | Buffers |
------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |          |      1 |        |     81 |       3 |
|   1 |  SORT ORDER BY         |          |      1 |     81 |     81 |       3 |
|   2 |   MAT_VIEW ACCESS FULL| SALES_MV |      1 |     81 |     81 |       3 |
------------------------------------------------------------------------------------
```

注意逻辑读的数值，与原始查询相比，从3094下降到了3。还要注意MAT_VIEW ACCESS FULL这个访问路径，它清楚地阐述了对物化视图的访问。这个访问路径操作起来就像全表扫描一样。这仅仅是一种用于方便指明使用了物化视图的命名约定。而实际上，这两种访问路径是完全一样的。

直接引用容器表始终是一个可选项。但是，如果想在不修改应用程序执行的SQL语句的情况下改进它的性能，则存在第二种有效的途径：使用查询重写。

注意　查询重写仅在企业版中可用。

查询重写的概念相当直白。当查询优化器接收到需要优化的查询，它可以决定按照原来的方式使用此查询（也就是说，不使用查询重写），或者它也可以选择重写此查询，以便使用包含执行该查询

所需的全部数据或部分数据的物化视图。图15-2演示了这一点。当然，这个决定是查询优化器分别在使用和不使用查询重写的情况下，根据对执行计划进行的成本估算做出的。拥有更低成本的执行计划会用于执行该查询。rewrite和no_rewrite这两个hint可以用于控制查询优化器的决定。

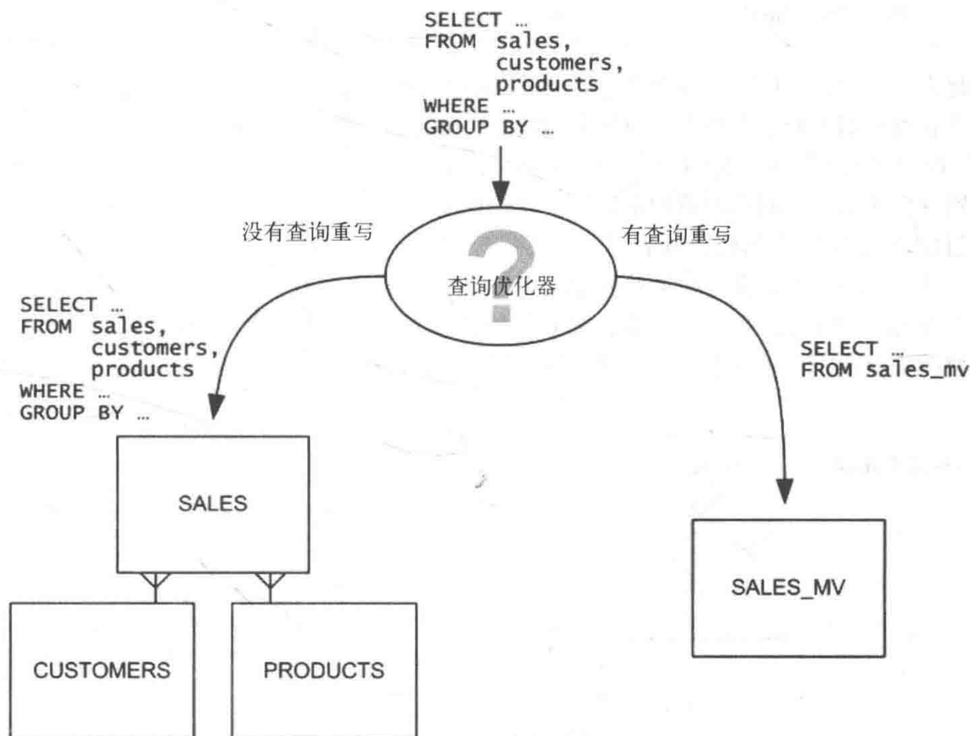


图15-2　查询优化器能够使用查询重写来自动利用物化视图

要利用查询重写，必须在两个级别上启用它。首先，必须将query_rewrite_enabled初始化参数设置为TRUE。其次，必须为物化视图启用此功能。下面的SQL语句展示如何为已经存在的物化视图启用查询重写：

```
ALTER MATERIALIZED VIEW sales_mv ENABLE QUERY REWRITE
```

如果启用了查询重写，那么一旦提交了原始的查询，查询优化器就会将此物化视图作为查询重写的候选项。在此情况下，查询优化器所做的，实际上是重写查询以便使用物化视图。注意，MAT_VIEW REWRITE ACCESS FULL明确地指出了查询重写的发生：

```
SELECT p.prod_category, c.country_id,
       sum(quantity_sold) AS quantity_sold,
       sum(amount_sold) AS amount_sold
FROM sales s, customers c, products p
WHERE s.cust_id = c.cust_id
AND s.prod_id = p.prod_id
GROUP BY p.prod_category, c.country_id
ORDER BY p.prod_category, c.country_id
```

```
-----------------------------------------------------------------------------------------
| Id | Operation                       | Name     | Starts | E-Rows | A-Rows | Buffers |
-----------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT                |          |      1 |        |     81 |       3 |
|  1 |  SORT ORDER BY                  |          |      1 |     81 |     81 |       3 |
|  2 |   MAT_VIEW REWRITE ACCESS FULL| SALES_MV |      1 |     81 |     81 |       3 |
-----------------------------------------------------------------------------------------
```

概括起来，通过查询重写，查询优化器能够自动使用包含执行一个查询所需数据的物化视图。打个比方，这有点类似于向表添加索引时发生的情况。你（通常）不必修改SQL语句去利用它。多亏了数据字典，查询优化器知道这样的一个索引存在，且如果它对于提升一条SQL语句的执行效率有帮助，查询优化器就会使用它。同样的道理也适用于物化视图。

当通过DML或DDL语句修改基表时，物化视图（实际上是容器表）可能会包含陈旧的数据（“陈旧”意为“老的”，也就是说，如果此时基于基表的新内容执行查询，数据已经不再等价于物化视图查询的结果集）。因此，数据库引擎会停止将物化视图用于查询重写。出于这个原因，如图15-3所示，在修改完基表后，必须执行物化视图的刷新。可以选择什么时间以哪种方式执行物化视图的刷新。

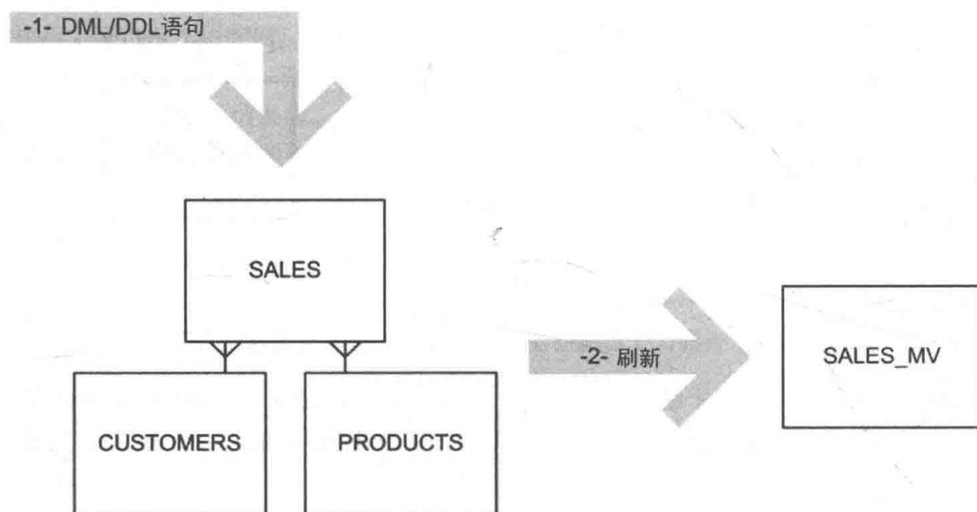


图15-3 在修改完基表之后，必须要刷新物化视图

现在我已经介绍了基本的概念，接下来会以更详细的方式，描述一下在创建物化视图期间可以指定哪些参数，以及查询重写和刷新是如何运作的。

2. 参数

正如上节中所述，在创建一张物化视图时可以不指定任何参数，不过也可以完全定制它的创建。

- 可以指定物理属性，比如分区、压缩、表空间等，也可以为容器表指定存储参数。就这一点而言，容器表会得到与所有其他堆表一样的对待。鉴于此，可以应用第13章中讨论的技术来进一步优化数据访问。

- 创建出物化视图之后，就会执行查询，然后会将结果集插入到容器表中。这是因为默认使用了build immediate参数。还存在两种额外的可选项：第一种，通过指定build deferred参数将数据的插入推迟到第一次刷新的时候；第二种，通过指定prebuilt table参数来重用一张已经存在的表作为容器表。
- 默认情况下，查询重写是禁用的。要启用它，必须指定enable query rewrite参数。只有企业版才支持启用查询重写。
- 为改进快速刷新（在本章稍后讲述）的性能，默认会在容器表上创建一个索引。要禁止这个索引的创建，可以指定using no index参数。这一点很有用，例如，为了避免索引维护的负载，而且这个负载绝对不可忽视，当然前提是你永远都不想执行快速刷新。

下面的SQL语句展示了基于之前同一个查询的例子，但是加入了几个刚刚描述过的参数：

```
CREATE MATERIALIZED VIEW sales_mv
PARTITION BY HASH (country_id) PARTITIONS 8
TABLESPACE users
BUILD IMMEDIATE
USING NO INDEX
ENABLE QUERY REWRITE
AS
SELECT p.prod_category, c.country_id,
       sum(quantity_sold) AS quantity_sold,
       sum(amount_sold) AS amount_sold
FROM sales s, customers c, products p
WHERE s.cust_id = c.cust_id
AND s.prod_id = p.prod_id
GROUP BY p.prod_category, c.country_id
```

此外，也可以指定物化视图如何刷新。“物化视图刷新”一节提供了关于这个主题的详细信息。

3. 查询重写

无论何时一个SELECT子句出现在SQL语句中，查询优化器都能够利用查询重写，或者，更具体一点说，在以下几种情况中：

- SELECT ... FROM ...
- CREATE TABLE ... AS SELECT ... FROM ...
- INSERT INTO ... SELECT ... FROM ...
- 子查询

此外，就像之前所说，只有在满足两个要求时才会使用查询重写。第一，必须将query_rewrite_enabled初始化参数设置为TRUE（默认值）。第二，物化视图必须使用enable query rewrite参数创建。

一旦满足这些要求，每次查询优化器生成一个执行计划，它都会查找是否存在一张包含所需数据的物化视图可以用于重写一条SQL语句。为了这个目的，它使用以下三种方法之一。

- 全文本匹配查询重写：查询语句的文本传递给查询优化器，用来与每个可用的物化视图的查询语句中的文本进行比较。如果它们匹配，显然该物化视图包含需要的数据。注意这个比较不像数据库引擎通常所用的比较那么严格：它不区分大小写（除了字面值）并且忽略空格（例如，新行和制表符）和ORDER BY子句。

- 部分文本匹配查询重写：这种比较类似于全文本匹配查询重写中使用的比较。但是这种方式允许在SELECT子句中出现不一致。举例来说，如果物化视图存储了三个列，而其中只有两个被需要优化的查询语句引用，则物化视图包含所有需要的数据，因此就可以使用查询重写。
- 通用查询重写：为了找到匹配的物化视图，通用查询重写会做一个语义分析。出于这个目的，它会广泛使用约束和维度来推断基表中数据的语义关系。其目的是，即使传递给查询优化器的查询语句与匹配的物化视图的查询语句有很大不同，也可以应用查询重写。事实上，对于设计良好的物化视图来说，会经常用于重写许多（而且还可能是不同的）SQL语句。

维度

查询优化器使用数据字典中存储的约束来推断数据关系，以最大程度地利用通用查询重写。有时，其他一些非常有用的关系，却没有被约束覆盖到，这种关系存在于同一张表的列之间，或者甚至是不同的表之间。对于非规范化的表（例如像sh模式对象下的times表）尤为如此。为了向查询优化器提供这样的信息，可以使用**维度**（dimension）。因为它，可以使用**层级**（hierarchy）指定1:n的关系以及可以使用**属性**（attribute）指定1:1的关系。层级和属性都是基于**级别**（level）的，也就是，简单来说，一张表中的列。下面的SQL语句进行了演示：

```
CREATE DIMENSION times_dim
LEVEL day IS times.time_id
LEVEL month IS times.calendar_month_desc
LEVEL quarter IS times.calendar_quarter_desc
LEVEL year IS times.calendar_year
HIERARCHY cal_rollup (day CHILD OF month CHILD OF quarter CHILD OF year)
ATTRIBUTE day DETERMINES (day_name, day_number_in_month)
ATTRIBUTE month DETERMINES (calendar_month_number, calendar_month_name)
ATTRIBUTE quarter DETERMINES (calendar_quarter_number)
ATTRIBUTE year DETERMINES (days_in_cal_year)
```

关于维度的详细信息可以在*Oracle Database Data Warehousing Guide*手册中找到。

可以很迅速地应用全文本匹配和部分文本匹配查询重写。但因为它们的判定是基于简单的文本匹配，所以不是很灵活。因此，它们只能重写有限数量的查询。比较起来，通用查询重写则要强大得多。不足之处是应用这种技术的负载也要高很多。出于这个原因，查询优化器依据复杂性的递增次序应用这些方法（从而分解开销），直到找到匹配的物化视图。此处理过程在图15-4中作了演示。

接下来的例子来自mv_rewrite.sql脚本，展示了通用查询重写的实战操作。注意该查询类似于上一小节中用于定义sales_mv物化视图的那个查询。以下是五个不同点。

- SELECT子句不一样。但是注意，此物化视图包含所有必要的数据。
- FROM子句是使用更新的联接语法书写的（注意JOIN和ON关键字）。
- GROUP BY子句不一样。数据是在更少的列上聚合的（与物化视图的定义相比，缺少country_id列）。
- 指定了一个ORDER BY子句。
- 没有引用customers表。然而，多亏sales表上一个引用了customers表的有效外键约束，查询优

化器能够判断出省略这个联接没有什么损失（因此，此联接无法消除任何数据）。

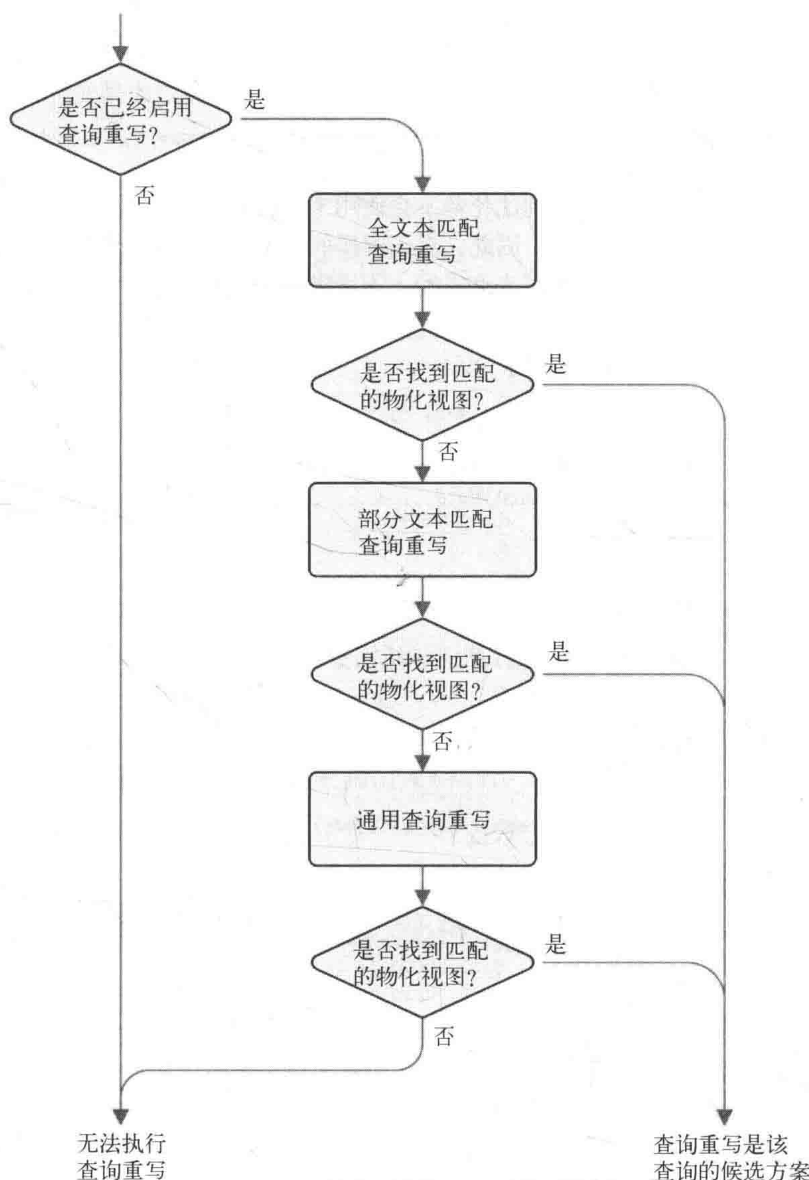


图15-4　查询重写处理过程

不管这些区别，使用通用查询重写，查询优化器能够利用sales_mv物化视图：

```
SQL> SELECT upper(p.prod_category) AS prod_category,
  2         sum(s.amount_sold) AS amount_sold
  3  FROM sales s JOIN products p ON s.prod_id = p.prod_id
  4  GROUP BY p.prod_category
  5  ORDER BY p.prod_category;
```

```
-----------------------------------------------------
| Id  | Operation                     | Name     |
-----------------------------------------------------
|   0 | SELECT STATEMENT              |          |
|   1 |  SORT GROUP BY                |          |
|   2 |   MAT_VIEW REWRITE ACCESS FULL| SALES_MV |
-----------------------------------------------------
```

值得注意的是，在默认情况下，查询优化器不会使用未验证的约束。因此，如果存在未验证的约束，查询优化器无法使用通用查询重写。因此，对于这样的查询来说，无法使用全文本匹配查询重写和部分文本匹配查询重写，所以不会出现查询重写。下面的例子证实了这一点。注意，除了FROM子句（但是，如之前所述，这个细节是无关的），这里使用的是与上一个例子中一样的查询。然而，sales_customer_fk约束的状态发生了改变：

```
SQL> ALTER TABLE sales MODIFY CONSTRAINT sales_customer_fk NOVALIDATE;

SQL> SELECT upper(p.prod_category) AS prod_category,
  2         sum(s.amount_sold) AS amount_sold
  3  FROM sales s, products p
  4  WHERE s.prod_id = p.prod_id
  5  GROUP BY p.prod_category
  6  ORDER BY p.prod_category;

---------------------------------------------
| Id  | Operation                  | Name     |
---------------------------------------------
|   0 | SELECT STATEMENT           |          |
|   1 |  SORT GROUP BY             |          |
|*  2 |   HASH JOIN                |          |
|   3 |    VIEW                    | VW_GBC_5 |
|   4 |     HASH GROUP BY          |          |
|   5 |      PARTITION RANGE ALL|          |
|   6 |       TABLE ACCESS FULL    | SALES    |
|   7 |    TABLE ACCESS FULL       | PRODUCTS |
---------------------------------------------

   2 - access("ITEM_1"="P"."PROD_ID")
```

尤其是对于数据集市，使用那些尽管对于数据库引擎来讲是未验证的但是却已知满足数据的需求的约束很常见，这要归功于（仔细地）维护表数据的方式。同时，拥有尽管被数据库引擎认为是陈旧的但是对于重写查询来说却是安全的物化视图也很常见。

要在这样的情况下利用通用查询重写，可以使用query_rewrite_integrity初始化参数。通过它，可以指定是否仅可以使用强制的约束（因而，对数据库引擎来说是验证的），以及是否使用包含陈旧数据的物化视图。可以将该参数设置为以下三个值。

- enforced：仅会考虑将包含最新数据的物化视图用于查询重写。注意，始终会认为基于外部表的物化视图是陈旧的。此外，仅验证过的约束用于通用查询重写。这是默认值。
- trusted：仅会考虑将包含最新数据的物化视图用于查询重写。此外，对于通用查询重写来说，未验证过且使用rely标记的维度和约束是可信的。

- stale_tolerated：所有存在的物化视图，包括那些含有陈旧数据的，都会考虑用于查询重写。此外，对于通用查询重写来说，未验证过且使用rely标记的维度和约束是可信的。

下面的例子展示如何在不验证约束的情况下使用通用查询重写。正如展示的那样，会对约束使用rely标记，并且会将完整性级别设置为trusted：

```
SQL> ALTER TABLE sales MODIFY CONSTRAINT sales_customer_fk RELY;

SQL> ALTER SESSION SET query_rewrite_integrity = trusted;

SQL> SELECT upper(p.prod_category) AS prod_category,
  2         sum(s.amount_sold) AS amount_sold
  3  FROM sales s, products p
  4  WHERE s.prod_id = p.prod_id
  5  GROUP BY p.prod_category
  6  ORDER BY p.prod_category;

--------------------------------------------------
| Id  | Operation                      | Name     |
--------------------------------------------------
|   0 | SELECT STATEMENT               |          |
|   1 |  SORT GROUP BY                 |          |
|   2 |   MAT_VIEW REWRITE ACCESS FULL| SALES_MV |
--------------------------------------------------
```

如果因为其中一个SQL语句没有使用查询重写而遇到麻烦，而且你不知道为什么，则可以使用dbms_mview包的explain_rewrite过程来找出问题所在。下面的PL/SQL代码块是一个如何使用它的例子。注意，query参数指定应该被重写的查询，mv参数指定应该用于重写的物化视图，而statement_id参数指定一个任意的字符串用于识别存储在输出表rewrite_table中的信息：

```
SQL> ALTER SESSION SET query_rewrite_integrity = enforced;

SQL> DECLARE
  2    l_query CLOB := 'SELECT upper(p.prod_category) AS prod_category,
  3                            sum(s.amount_sold) AS amount_sold
  4                     FROM sh.sales s, sh.products p
  5                     WHERE s.prod_id = p.prod_id
  6                     GROUP BY p.prod_category
  7                     ORDER BY p.prod_category';
  8  BEGIN
  9    DELETE rewrite_table WHERE statement_id = '42';
 10    dbms_mview.explain_rewrite(
 11      query        => l_query,
 12      mv           => 'sh.sales_mv',
 13      statement_id => '42'
 14    );
 15  END;
 16  /
```

注意　rewrite_table表默认情况下并不存在。你可以登录到用于分析的模式对象下，通过执行存储在$ORACLE_HOME/rdbms/admin目录下的utlxrw.sql脚本来创建它。

该过程的输出，在rewrite_table表中提供，会给出为什么查询重写没有发生的原因。输出由 *Oracle Database Error Messages* 手册中记录的信息组成。以下是之前的分析的输出：

```
SQL> SELECT message
  2  FROM rewrite_table
  3  WHERE statement_id = '42';

MESSAGE
-------------------------------------------------------------------------------------------
QSM-01150: query did not rewrite
QSM-01284: materialized view SALES_MV has an anchor table CUSTOMERS not found in query
QSM-01052: referential integrity constraint on table, SALES, not VALID in ENFORCED integrity
mode
```

还需要注意的是，并非所有查询重写方法都可以应用到所有物化视图上。某些物化视图只支持全文本匹配查询重写。其他的一些只支持全文本匹配和部分文本匹配查询重写。总的来说，随着物化视图复杂性（例如，考虑到像集合运算符和层次查询的使用）的增加，对于高级查询重写的支持就越少。限制条件还取决于Oracle数据库版本。所以，我不会提供一个受支持查询重写的清单，取而代之的是，我会向你展示，对于一个给定的例子，如何找出哪些查询重写方法是受支持的。为了演示，我们使用下面的SQL语句重新创建物化视图。注意，与之前的例子相比，我只是将p.prod_status 添加到了GROUP BY子句中（在实践中，执行这样的一个SQL语句通常没什么意义，但是，很快你就会发现，这是使查询重写部分失效的一种相对简单的方式）：

```
CREATE MATERIALIZED VIEW sales_mv
ENABLE QUERY REWRITE
AS
SELECT p.prod_category, c.country_id,
       sum(s.quantity_sold) AS quantity_sold,
       sum(s.amount_sold) AS amount_sold
FROM sales s, customers c, products p
WHERE s.cust_id = c.cust_id
AND s.prod_id = p.prod_id
GROUP BY p.prod_category, c.country_id, p.prod_status
```

要显示一个物化视图支持的查询重写方法，可以查询下面例子中所示的user_mviews视图（也可以使用相应的dba、all以及在12.1版本中的多租户环境下的cdb版本的视图）。在这个案例中，根据rewrite_enabled列，查询重写在该物化视图级别上是启用的，而且根据rewrite_capability列，只有文本匹配查询重写是受支持的（换句话说，不支持通用查询重写）：

```
SQL> SELECT rewrite_enabled, rewrite_capability
  2  FROM user_mviews
  3  WHERE mview_name = 'SALES_MV';

REWRITE_ENABLED REWRITE_CAPABILITY
--------------- ------------------
Y               TEXTMATCH
```

注意，rewrite_capability列只能取以下这些值之一：none、textmatch或general。如果支持通用查询重写（因此也支持其他两种方法），则user_mviews视图提供的信息就足够了。然而，如此案例中所示，如果显示的是值textmatch，则还至少需要知道另外两件事情。第一，在这两种文本匹配查询重写方法中，哪一种是受支持的？是只支持全文本匹配查询重写，还是也支持部分文本匹配查询重写？第二，为什么不支持通用查询重写？

要回答这些问题，可以使用dbms_mview包中的explain_mview过程，如下例所示。注意mv参数指定物化视图的名称，stmt_id参数指定一个任意的字符串用于识别存储在输出表mv_capabilities_table中的信息：

```
SQL> execute dbms_mview.explain_mview(mv => 'sales_mv', stmt_id => '42')
```

注意 mv_capabilities_table表默认情况下并不存在。可以通过执行在$ORACLE_HOME/rdbms/admin目录下存储的utlxmv.sql脚本，在要使用它的模式对象下创建此表。

可以在mv_capabilities_table表中找到该过程的输出，该输出展示了sales_mv物化视图是否支持这三种查询重写模式。如果不支持，msgtxt列会表明某个具体的查询重写模式因为什么原因不受支持。在这个案例中你会注意到，问题只是由于缺少一个在GROUP BY子句中引用的列引起的（向上检查此SQL语句，你会立即发现引起问题的列：p.prod_status）：

```
SQL> SELECT capability_name, possible, msgtxt
  2  FROM mv_capabilities_table
  3  WHERE statement_id = '42'
  4  AND capability_name IN ('REWRITE_FULL_TEXT_MATCH',
  5                          'REWRITE_PARTIAL_TEXT_MATCH',
  6                          'REWRITE_GENERAL');
CAPABILITY_NAME            POSSIBLE MSGTXT
-------------------------- -------- ----------------------------------------
REWRITE_FULL_TEXT_MATCH    Y
REWRITE_PARTIAL_TEXT_MATCH N        grouping column omitted from SELECT list
REWRITE_GENERAL            N        grouping column omitted from SELECT list
```

4. 刷新

修改了某张表之后，所有依赖的物化视图都会变得陈旧。因此，对于每一个陈旧的物化视图，有必要进行刷新。可以在创建物化视图时，指定刷新如何以及何时发生。

要指定数据库引擎如何执行刷新，可以从以下方法中选择。

- REFRESH COMPLETE：容器表中的全部内容都会被删除，所有数据都会从基表重新加载。显然，这种方法总是受支持的。只有当基表中相当大的一部分被修改时，或由于物化视图的复杂性快速刷新不可用时，才应该使用这种方法。
- REFRESH FAST：容器表中的内容被重复利用，只有修改的部分会被传播至容器表。如果基表中的少量数据被修改，应该使用这种方法。仅当满足几个要求的情况下这种方法才可用。如果其中的一个没有满足，要么会拒绝将REFRESH FAST作为物化视图的一个合法参数，要么会引发一个错误。快速刷新和PCT刷新（一种特别的快速刷新），都会在下一小节中详细介绍。

- REFRESH FORCE：起初，尝试进行快速刷新。如果不起作用，就会执行完整刷新。这是默认的方法。
- NEVER REFRESH：物化视图从不进行刷新。如果尝试进行刷新，会随着ORA-23538: cannot explicitly refresh a NEVER REFRESH materialized view错误终止。可以使用这种方法来确保永远不会执行刷新。

可以通过以下两种方式选择物化视图刷新出现的时间点。

- ON DEMAND：物化视图只有在被明确要求的情况下刷新（或者是手动或者是通过按固定间隔运行一个任务）。这意味着在从基表发生修改到物化视图刷新的这段时间内，物化视图可能会包含陈旧的数据。
- ON COMMIT：物化视图会在修改它引用的基表的事务结束时自动刷新。换句话说，就其他的会话而言，物化视图总是包含最新的数据。

可以组合这些选项来指定物化视图如何刷新，以及何时进行刷新，并且在CREATE MATERIALIZED VIEW和ALTER MATERIALIZED语句中都可以使用它们。下面是一个例子：

```
ALTER MATERIALIZED VIEW sales_mv REFRESH FORCE ON DEMAND
```

你甚至可以使用REFRESH COMPLETE ON COMMIT选项创建一张物化视图。然而，这样的配置好像不太可能在实践中有什么用处。

要显示与一个物化视图有关的参数，它是否是最新的，还有上一次刷新是如何以及何时操作的，可以查询user_mviews视图（也可以使用相关的dba、all以及在12.1版本中的多租户环境下的cdb版本）。

```
SQL> SELECT refresh_method, refresh_mode, staleness, last_refresh_type, last_refresh_date
  2  FROM user_mviews
  3  WHERE mview_name = 'SALES_MV';

REFRESH_METHOD REFRESH_MODE STALENESS LAST_REFRESH_TYPE LAST_REFRESH_DATE
-------------- ------------ --------- ----------------- -----------------
FORCE          DEMAND       FRESH     COMPLETE          2013-12-10 15:51
```

如果选择手动刷新物化视图，可以调用dbms_mview包中的以下过程之一。

- refresh：此过程刷新通过list参数指定为逗号分隔列表的物化视图。例如，以下调用会刷新sh用户拥有的sales_mv和cal_month_sales_mv物化视图：

```
dbms_mview.refresh(list => 'sh.sales_mv,sh.cal_month_sales_mv')
```

- refresh_all_mviews：除了那些标记为永不刷新的物化视图以外，此过程刷新在数据库中存储的所有物化视图。输出参数number_of_failures返回在处理期间出现的失败的数量：

```
dbms_mview.refresh_all_mviews(number_of_failures => :r)
```

- refresh_dependent：此过程刷新某种物化视图，这种物化视图依赖通过list参数指定为逗号分隔列表的基表。输出参数number_of_failures返回在处理期间出现的失败的数量。例如，以下调用会根据sh用户拥有的sales表，刷新所有物化视图：

```
dbms_mview.refresh_dependent(number_of_failures => :r, list => 'sh.sales')
```

所有这些过程还支持参数method和atomic_refresh。前者指定如何完成刷新（'c'代表完全刷新，'f'代表快速刷新，'p'代表PCT刷新，'?'代表强制），后者指定刷新是否在一个单独的事务中执行。

如果将atomic_refresh参数设置为FALSE（默认值是TRUE），则不使用单独的事务。结果就是，对于完全刷新，物化视图是被截断而不是被删除。一方面，刷新更快速了。另一方面，如果另一个会话在刷新运行期间查询物化视图，该查询可能会返回错误的结果（没有选中任何数据）。

此外，从12.1版本开始，一个称作out_of_place的新参数可用了。如果将这个out_of_place参数设置为FALSE（默认值），刷新会直接在与物化视图关联的容器表上执行。这样的刷新方式对于访问该物化视图的并发查询来讲可能会导致一些问题。

如果将out_of_place设置为TRUE，那么刷新会在另一张表的帮助下执行。实际发生的是，会创建出另外一张容器表，会通过直接路径插入将最新的数据插入到这张表中，新的容器表与旧的容器表进行切换，最终丢弃掉旧的容器表。这种方法，称作out-of-place刷新，确保将并发查询访问物化视图的影响降到最低。缺点是在刷新期间，需要的空间加倍。

一旦想要按需自动化一个刷新，通过CREATE MATERIALIZED VIEW和ALTER MATERIALIZED VIEW，都可以指定第一次刷新的时间（START WITH子句）和一个表达式来评估随后的刷新的时间（NEXT子句）。例如，使用下面的SQL语句，会将刷新安排在执行SQL语句时开始，每十分钟进行一次：

```
ALTER MATERIALIZED VIEW sales_mv
REFRESH COMPLETE ON DEMAND
START WITH sysdate
NEXT sysdate+to_dsinterval('0 00:10:00')
```

为调度该刷新，会自动提交一个基于dbms_job包的后台任务。注意dbms_refresh包用于代替dbms_mview包：

```
SQL> SELECT what, interval
  2  FROM user_jobs;

WHAT                                       INTERVAL
------------------------------------------ -----------------------------------
dbms_refresh.refresh('"CHRIS"."SALES_MV"'); sysdate+to_dsinterval('0 00:10:00')
```

刷新组

dbms_refresh包用于管理**刷新组**（REFRESH GROUP）。一个刷新组是由一个或多个物化视图组成的一个简单集合。通过dbms_refresh包中的刷新过程执行的刷新是在一个单独的事务中执行的（atomic_refresh被设置为TRUE）。如果需要保证几个物化视图之间的一致性，那么这种行为是有必要的。这还意味着要么会成功刷新包含在组中的所有物化视图，要么会回滚整个刷新。

- **使用物化视图日志进行快速刷新**

在快速刷新期间，会重复利用容器表的内容，且仅会将修改的部分从基表传播至容器表中。显然，数据库引擎只有在知道修改的部分时才能够传播它们。出于这个目的，必须在每个基表上创建一个物化视图日志（materialized view log）才能启用快速刷新（分区变化跟踪快速刷新会在下一小节中讨论，是现在讨论内容的一个例外）。例如，为了能够快速刷新，sales_mv物化视图需要sales、customers和products表上的物化视图日志。

简而言之，物化视图日志是一张由数据库引擎自动维护的表，用于跟踪出现在基表上的变更。除

了物化视图日志之外，还有一张内部日志表用于直接路径插入。你不需要创建它，因为它是在数据库创建的时候自动安装的。要显示它的内容，可以查询all_sumdelta视图。

在最简单的案例中，可以使用像下面这样的SQL语句创建物化视图日志（这个例子来自mv_refresh_log.sql脚本）：

```
SQL> CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID;

SQL> CREATE MATERIALIZED VIEW LOG ON customers WITH ROWID;

SQL> CREATE MATERIALIZED VIEW LOG ON products WITH ROWID;
```

可以添加WITH ROWID子句来指定如何识别物化视图日志中的记录，也就是指定如何识别那些被每个物化视图日志行跟踪的修改的基表记录。也可以创建使用主键或对象ID来识别修改的记录的物化视图日志。但是，针对本章的目的，记录必须通过它们的rowid进行识别（其他的方式适合在分布式环境中使用物化视图的时候使用）。

就像物化视图有一张关联的容器表一样，每个物化视图日志也有一张关联的表，用于记录对基表所做的修改。下面的查询展示如何显示它的名称：

```
SQL> SELECT master, log_table
  2  FROM dba_mview_logs
  3  WHERE master IN ('SALES', 'CUSTOMERS', 'PRODUCTS')
  4  AND log_owner = 'SH';

MASTER    LOG_TABLE
--------- ---------------
CUSTOMERS MLOG$_CUSTOMERS
PRODUCTS  MLOG$_PRODUCTS
SALES     MLOG$_SALES
```

对于某些物化视图，这样一个基本的物化视图日志并不足以支持快速刷新。有额外的要求需要得到满足。因为这些要求强烈依赖于与物化视图有关的查询以及Oracle数据库版本，我不会提供一个列表，而是会向你展示对于一个给定的案例，如何找出这些要求是什么。为此，你可以使用在查找有哪些支持的查询重写模式时使用的相同方法（见15.1.1的“查询重写”一节）。换句话说，可以使用dbms_mview包中的explain_mview过程，如下面的例子展示的这样：

```
SQL> execute dbms_mview.explain_mview(mv => 'sales_mv', stmt_id => '42')
```

该过程的输出在mv_capabilities_table表中提供。要查看是否可以快速刷新物化视图，可以使用类似下面这样的查询。注意在输出中，总是会将possible这一列设置为N。它的意思是没有可行的快速刷新。此外，msgtxt和related_text列表明了这个问题的原因：

```
SQL> SELECT capability_name, possible, msgtxt, related_text
  2  FROM mv_capabilities_table
  3  WHERE statement_id = '42'
  4  AND capability_name LIKE 'REFRESH_FAST_AFTER%';

CAPABILITY_NAME              POSSIBLE MSGTXT                          RELATED_TEXT
---------------------------- -------- ------------------------------- -------------
REFRESH_FAST_AFTER_INSERT    N        mv log must have new values     SH.PRODUCTS
REFRESH_FAST_AFTER_INSERT    N        mv log does not have all necess SH.PRODUCTS
```

```
                                             ary columns
REFRESH_FAST_AFTER_INSERT     N              mv log must have new values     SH.CUSTOMERS
REFRESH_FAST_AFTER_INSERT     N              mv log does not have all necess SH.CUSTOMERS
                                             ary columns
REFRESH_FAST_AFTER_INSERT     N              mv log must have new values     SH.SALES
REFRESH_FAST_AFTER_INSERT     N              mv log does not have all necess SH.SALES
                                             ary columns
REFRESH_FAST_AFTER_ONETAB_DML N              SUM(expr) without COUNT(expr)   AMOUNT_SOLD
REFRESH_FAST_AFTER_ONETAB_DML N              SUM(expr) without COUNT(expr)   QUANTITY_SOLD
REFRESH_FAST_AFTER_ONETAB_DML N              see the reason why REFRESH_FAST
                                             _AFTER_INSERT is disabled
REFRESH_FAST_AFTER_ONETAB_DML N              COUNT(*) is not present in the
                                             select list
REFRESH_FAST_AFTER_ONETAB_DML N              SUM(expr) without COUNT(expr)
REFRESH_FAST_AFTER_ANY_DML    N              mv log does not have sequence # SH.PRODUCTS
REFRESH_FAST_AFTER_ANY_DML    N              mv log does not have sequence # SH.CUSTOMERS
REFRESH_FAST_AFTER_ANY_DML    N              mv log does not have sequence # SH.SALES
REFRESH_FAST_AFTER_ANY_DML    N              see the reason why REFRESH_FAST
                                             _AFTER_ONETAB_DML is disabled
```

其中的一些问题和物化视图日志有关，其他的则与物化视图有关。简而言之，数据库引擎需要更多的信息来执行快速刷新。

为了解决与物化视图日志有关的问题，必须在CREATE MATERIALIZED VIEW LOG语句中添加一些选项。

- 对于“mv log does not have all necessary columns”问题，必须指定每个被物化视图引用的列都存储在物化视图日志中。
- 对于“mv log must have new values”问题，必须添加INCLUDING NEW VALUES子句。通过这个选项，物化视图日志会在执行更新被时将旧值和新值（默认情况下仅会存储旧值）都存储起来（也就是说，会有两条记录写入到物化视图日志中）。
- 对于“mv log does not have sequence”问题，则有必要添加SEQUENCE子句。通过这个选项，会有一个序列号与物化视图日志中存储的每一行数据相关联。

下面是重新定义的物化视图日志：

```
SQL> CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID, SEQUENCE
  2  (cust_id, prod_id, quantity_sold, amount_sold) INCLUDING NEW VALUES;

SQL> CREATE MATERIALIZED VIEW LOG ON customers WITH ROWID, SEQUENCE
  2  (cust_id, country_id) INCLUDING NEW VALUES;

SQL> CREATE MATERIALIZED VIEW LOG ON products WITH ROWID, SEQUENCE
  2  (prod_id, prod_category) INCLUDING NEW VALUES;
```

TIMESTAMP与COMMIT SCN-BASED物化视图日志

从12.1版本开始，有两种类型的物化视图日志：基于时间戳的和基于提交SCN号的。因为基于时间戳的物化视图日志是在11.1.0.7及之前的版本中唯一存在的形式，所以在后续的版本中也是默认使用它们。要使用新的类型，必须在创建物化视图日志时指定COMMIT SCN子句。下面的SQL语句展示了一个例子：

```
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID, COMMIT SCN, SEQUENCE (cust_id, prod_id,
quantity_sold, amount_sold) INCLUDING NEW VALUES
```

尽管从用户的角度出发，这两种类型之间没有区别，但是物化视图日志所使用的快速刷新的算法，在基于提交SCN号时会带来更好的性能。

注意基于提交SCN号的物化视图日志默认没有启用，因为它们受到一些限制。例如，带有LOB列的表是不受支持的。

为了解决与物化视图相关的问题，必须将一些基于count函数的新列添加到与物化视图关联的查询中。下面的SQL语句展示了包含新列的定义：

```
CREATE MATERIALIZED VIEW sales_mv
REFRESH FORCE ON DEMAND
AS
SELECT p.prod_category, c.country_id,
       sum(s.quantity_sold) AS quantity_sold,
       sum(s.amount_sold) AS amount_sold,
       count(*) AS count_star,
       count(s.quantity_sold) AS count_quantity_sold,
       count(s.amount_sold) AS count_amount_sold
FROM sales s, customers c, products p
WHERE s.cust_id = c.cust_id
AND s.prod_id = p.prod_id
GROUP BY p.prod_category, c.country_id
```

在重新定义完物化视图日志和物化视图之后，在使用explain_mview过程的进一步分析中，结果显示快速刷新在所有情况下都是可行的（possible列都设置为Y）。那么我们通过向两张表中插入数据，然后执行快速刷新来测试一下刷新到底有多快：

```
SQL> INSERT INTO products
  2  SELECT 619, prod_name, prod_desc, prod_subcategory, prod_subcategory_id,
  3         prod_subcategory_desc, prod_category, prod_category_id,
  4         prod_category_desc, prod_weight_class, prod_unit_of_measure,
  5         prod_pack_size, supplier_id, prod_status, prod_list_price,
  6         prod_min_price, prod_total, prod_total_id, prod_src_id,
  7         prod_eff_from, prod_eff_to, prod_valid
  8  FROM products
  9  WHERE prod_id = 136;

SQL> INSERT INTO sales
  2  SELECT 619, cust_id, time_id, channel_id, promo_id, quantity_sold, amount_sold
  3  FROM sales
  4  WHERE prod_id = 136;

SQL> COMMIT;

SQL> execute dbms_mview.refresh(list => 'sh.sales_mv', method => 'f')

Elapsed: 00:00:00.12
```

在本例中，快速刷新持续了0.12秒钟。如果不满意快速刷新的性能，应该使用SQL跟踪调查为何

它花费了这么久。然后，通过应用第13章中描述的技巧，你可能能够通过添加索引（在主表上，在物化视图上，或甚至有时候在物化视图日志上添加）或给一个物理段分区来加速查询。

提示 根据你使用的版本以及你遇到问题的物化视图的类型，有几个未公开的参数有可能会影响性能。讨论这些参数超出了本书的范围。如果有关于快速刷新的性能问题，建议查看一下Oracle Support文档*Master Note for Materialized View*（1353040.1），特别是“Performance Issues with MVIEW”一节。

● **使用分区变化跟踪进行快速刷新**

存储历史数据的表经常会按照日、星期或年进行范围分区。换句话说，分区是基于存储时间信息的列。因此，新分区的加入，数据加载到分区中，以及丢弃旧分区（通常只会保持特定数量的分区在线）都是有规律地发生的。在执行完这些操作后，所有依赖的物化视图都是陈旧的，并且因此应该进行刷新。

问题是使用物化视图日志的快速刷新（如上一小节所描述），无法在类似ADD PARTITION或DROP PARTITION这样的分区管理操作之后执行。如果尝试这样的刷新，数据库引擎会引发一个ORA-32313: REFRESH FAST of <mview> unsupported after PMOPs错误。当然了，完全刷新总是可以执行的。然而，如果有很多的分区而且只有其中一个或两个被修改了，完全刷新的时间可能是不可接受的。

要解决这个问题，可以使用基于分区变化跟踪（partition change tracking，PCT）的快速刷新。这样做可行是因为，数据库引擎不仅仅在表级别而且在分区级别也有能力跟踪数据是否陈旧。换句话说，对于所有没有修改过的分区，可以跳过刷新。要使用这种刷新方法，物化视图必须满足一些要求。基本上就是，数据库引擎必须能够将物化视图中存储的数据映射到基表的分区上。如果物化视图包含以下各项之一，这就是可行的：

- ❑ 分区键
- ❑ Rowid
- ❑ 分区标记（partition marker）
- ❑ 联接相关（Join-dependent）表达式

前两个是什么应该是显而易见的；我们来看一下第三个和第四个的例子。一个分区标记只不过是由dbms_mview包中的pmarker函数生成的一个分区标识符（实际上，它是与分区段有关的数据对象ID）。要生成分区标记，此函数使用rowid作为一个参数传递进来。下面的例子来自mv_refresh_pct.sql脚本，展示了如何创建包含分区标记的物化视图（注意，sales表是分区的）：

```
CREATE MATERIALIZED VIEW sales_mv
REFRESH FORCE ON DEMAND
AS
SELECT p.prod_category, c.country_id,
       sum(quantity_sold) AS quantity_sold,
       sum(amount_sold) AS amount_sold,
       count(*) AS count_star,
       count(quantity_sold) AS count_quantity_sold,
       count(amount_sold) AS count_amount_sold,
       dbms_mview.pmarker(s.rowid) AS pmarker
```

```
FROM sales s, customers c, products p
WHERE s.cust_id = c.cust_id
AND s.prod_id = p.prod_id
GROUP BY p.prod_category, c.country_id, dbms_mview.pmarker(s.rowid)
```

注意 因为每一行都要调用pmarker函数，不要低估此调用所需的时间。在我的系统上，创建包含分区标记的物化视图花费的时间比不使用分区标记的时间长2.5倍。

当物化视图在SELECT子句中引用的其中一个列，来自一张通过基于分区键进行等值谓词联接的表时，该物化视图包含一个联接相关表达式。在本节使用的例子中，它的意思是不仅会将与times表相关的等值联接（s.time_id = t.time_id）加入到物化视图中，而且会将times表的其中一个列（t.fiscal_year）添加到SELECT和GROUP BY子句中。下面是一个例子：

```
CREATE MATERIALIZED VIEW sales_mv
REFRESH FORCE ON DEMAND
AS
SELECT p.prod_category, c.country_id, t.fiscal_year,
       sum(quantity_sold) AS quantity_sold,
       sum(amount_sold) AS amount_sold,
       count(*) AS count_star,
       count(quantity_sold) AS count_quantity_sold,
       count(amount_sold) AS count_amount_sold
FROM sales s, customers c, products p, times t
WHERE s.cust_id = c.cust_id
AND s.prod_id = p.prod_id
AND s.time_id = t.time_id
GROUP BY p.prod_category, c.country_id, t.fiscal_year
```

使用了分区标记或分区相关表达式后，基于dbms_mview包中explain_mview函数的一个分析告诉我们基于分区变化跟踪的快速刷新是可行的。然而，它只对于在sales表上执行的修改是可行的：

```
SQL> SELECT capability_name, possible, msgtxt, related_text
  2  FROM mv_capabilities_table
  3  WHERE statement_id = '43'
  4  AND capability_name IN ('PCT_TABLE','REFRESH_FAST_PCT');

CAPABILITY_NAME  POSSIBLE MSGTXT                             RELATED_TEXT
---------------- -------- ---------------------------------- ------------
PCT_TABLE        Y                                           SALES
PCT_TABLE        N        relation is not a partitioned table CUSTOMERS
PCT_TABLE        N        relation is not a partitioned table PRODUCTS
REFRESH_FAST_PCT Y
```

15.1.2 何时使用

物化视图是冗余的访问结构。与所有冗余访问结构一样，它们对于提高访问数据的效率有帮助，但是它们会为了保持最新而增加负载。如果将物化视图与索引进行对比，物化视图带来的效率提升和负载可能都会比索引高出很多。很明显，这两个概念旨在解决不同的问题。简而言之，只有在加速数

据访问的正面作用超过了管理冗余数据副本（例如索引）的负面作用时，才应该使用物化视图。

总的来说，我觉得物化视图有以下两个作用。

- 在逻辑读数量和返回记录数量之间的比值非常高的时候，用于改进大型聚合或联接操作的性能。
- 在全表扫描和索引扫描效率都不高的时候，用于改进单表访问的性能。基本上，如果这些访问拥有平均选择率则可能会需要分区，但是如果无法利用分区（第13章讨论过什么时候这是不可行的），则物化视图可能会有所帮助。

通常可以在数据仓库中使用物化视图构建存储聚合。这样做有两个原因。第一，数据大部分是只读的；因此，当数据库仅致力于修改表的时候，可以将刷新物化视图的负载最小化并通过时间窗口隔离开来。第二，在这样的环境中，性能提升可能是巨大的。事实上，不使用物化视图时，经常会发现基于大型聚合或联接的查询会请求无法接受的需要处理的资源总量。

即使数据仓库是使用物化视图的主要环境，我也曾经在OLTP系统上成功地实施过它们。这可能对那些经常被查询而相对而言却很少进行修改的表有利。在这样的环境中刷新物化视图时，经常使用on commit快速刷新，以便保证亚秒级刷新并时刻保持最新的物化视图。

15.1.3 陷阱和谬误

因为快速刷新并非总是比完全刷新快，所以并非在所有的情况下都应该使用快速刷新。具体来讲，比如在基表中有大量数据被修改的情况。此外，也不要低估在修改基表的同时维护物化视图日志带来的负载。因此，应该仔细评估使用快速刷新的利弊。

当创建物化视图日志时，对于逗号的使用必须非常小心。看出下面的SQL语句哪里出问题了么？

```
SQL> CREATE MATERIALIZED VIEW LOG ON products WITH ROWID, SEQUENCE,
  2  (prod_id, prod_category) INCLUDING NEW VALUES;
CREATE MATERIALIZED VIEW LOG ON products WITH ROWID, SEQUENCE,
*
ERROR at line 1:
ORA-12026: invalid filter column detected
```

问题出在关键字SEQUENCE和过滤清单（换句话说，括号之间的列清单）之间的逗号。如果出现了逗号，则隐含PRIMARY KEY选项，而在本例中不可以指定该选项，因为主键（prod_id列）已经在过滤清单中了。下面是正确的SQL语句。注意，只是移除了逗号：

```
SQL> CREATE MATERIALIZED VIEW LOG ON products WITH ROWID, SEQUENCE
  2 (prod_id, prod_category) INCLUDING NEW VALUES;

Materialized view log created.
```

15.2 结果缓存

缓存是计算机系统改进性能所用的最常见技术之一。无论硬件还是软件都对其有着广泛的应用。Oracle数据库也不例外。举例来说，Oracle数据库在缓冲区缓存中缓存数据文件块，在数据字典缓存中缓存数据字典信息，以及在库缓存中缓存游标。从11.1版本开始，结果缓存也可用了。

注意　结果缓存仅在企业版中可用。

15.2.1　工作原理

Oracle数据库提供以下三种类型的结果缓存。

- 服务器结果缓存（也称作查询结果缓存）是存储查询结果集的服务器端缓存。
- PL/SQL函数结果缓存是存储PL/SQL函数返回值的服务器端缓存。
- 客户端结果缓存是存储查询结果集的客户端缓存。

接下来的小节描述这些缓存如何运作，以及要利用它们你必须要做哪些事情。记住，默认情况下并不会启用结果缓存。

1. 服务器结果缓存

服务器结果缓存可以用于避免查询和某些子查询(在WITH子句中定义的子查询和在FROM子句中定义的内联视图）的重复执行。简言之，当第一次执行某个查询时，它的结果集就会存储在共享池中。然后，对于后续执行的相同查询来说，结果集直接由结果缓存提供而不需要重新计算。注意如果认为两个查询是相等的，那么可以使用相同的缓存结果集，前提是它们拥有相同的文本（但是在空格和大小写方面的区别是允许的）。此外，如果有绑定变量出现，它们的值必须完全相同。这是有必要的，因为很明显绑定变量是传递给查询的输入参数，所以对于不同的绑定变量值，结果集通常也会不同。还要注意因结果缓存存储在共享池中，所以连接到一个给定数据库实例的所有会话共享相同的缓存条目。

为了向你展示一个例子（来自rc_query_hint.sql脚本），我们来执行已经在物化视图一节中使用了两次的查询（注意，在查询中指定了result_cache hint来启用结果缓存）：

```
SQL> SELECT /*+ result_cache */
  2         p.prod_category, c.country_id,
  3         sum(s.quantity_sold) AS quantity_sold,
  4         sum(s.amount_sold) AS amount_sold
  5  FROM sales s, customers c, products p
  6  WHERE s.cust_id = c.cust_id
  7  AND s.prod_id = p.prod_id
  8  GROUP BY p.prod_category, c.country_id
  9  ORDER BY p.prod_category, c.country_id;

Elapsed: 00:00:01.25

-----------------------------------------------------------------------------------
| Id  | Operation               | Name                       | Starts | A-Rows |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |                            |      1 |     81 |
|   1 |  RESULT CACHE           | 089x05gkvfuxq7wqg06u9z0zkb |      1 |     81 |
|   2 |   SORT GROUP BY         |                            |      1 |     81 |
|*  3 |    HASH JOIN            |                            |      1 |    956 |
|   4 |     TABLE ACCESS FULL   | PRODUCTS                   |      1 |     72 |
|   5 |     VIEW                | VW_GBC_9                   |      1 |    956 |
|   6 |      HASH GROUP BY      |                            |      1 |    956 |
```

```
|* 7 |       HASH JOIN          |                                |     1 |   918K|
|  8 |        TABLE ACCESS FULL | CUSTOMERS                      |     1 |  55500 |
|  9 |        PARTITION RANGE ALL|                               |     1 |   918K|
| 10 |         TABLE ACCESS FULL | SALES                          |    28 |   918K|
--------------------------------------------------------------------------------

   3 - access("ITEM_1"="P"."PROD_ID")
   7 - access("S"."CUST_ID"="C"."CUST_ID")
```

第一次执行花费了1.25秒。注意在执行计划中，RESULT CACHE操作已确认为这个查询启用了结果缓存。然而，执行计划的Starts列清楚地显示出所有操作都被执行了至少一次。所有操作的执行都是有必要的，因为这是该查询的第一次执行，也就是说，结果缓存还没有包含相应的结果集。

第二次执行变快了（0.16秒）：

```
SQL> SELECT /*+ result_cache */
  2         p.prod_category, c.country_id,
  3         sum(s.quantity_sold) AS quantity_sold,
  4         sum(s.amount_sold) AS amount_sold
  5  FROM sales s, customers c, products p
  6  WHERE s.cust_id = c.cust_id
  7  AND s.prod_id = p.prod_id
  8  GROUP BY p.prod_category, c.country_id
  9  ORDER BY p.prod_category, c.country_id;

Elapsed: 00:00:00.16

--------------------------------------------------------------------------------
| Id  | Operation                | Name                           | Starts | A-Rows |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |                                |      1 |     81 |
|   1 |  RESULT CACHE            | 089x05gkvfuxq7wqg06u9z0zkb     |      1 |     81 |
|   2 |   SORT GROUP BY          |                                |      0 |      0 |
|*  3 |    HASH JOIN             |                                |      0 |      0 |
|   4 |     TABLE ACCESS FULL    | PRODUCTS                       |      0 |      0 |
|   5 |     VIEW                 | VW_GBC_9                       |      0 |      0 |
|   6 |      HASH GROUP BY       |                                |      0 |      0 |
|*  7 |       HASH JOIN          |                                |      0 |      0 |
|   8 |        TABLE ACCESS FULL | CUSTOMERS                      |      0 |      0 |
|   9 |        PARTITION RANGE ALL|                               |      0 |      0 |
|  10 |         TABLE ACCESS FULL | SALES                          |      0 |      0 |
--------------------------------------------------------------------------------

   3 - access("ITEM_1"="P"."PROD_ID")
   7 - access("S"."CUST_ID"="C"."CUST_ID")
```

这一次，执行计划中的Starts列显示除了RESULT CACHE之外没有执行其他操作。注意第10章中提到过的一个行源操作可以完全避免调用它的子操作，因为它不需要子操作来完成它的工作，这正是那些案例之一。换句话说，该查询的结果集直接从结果缓存中提供。

在这个执行计划中，有意思的是，我们注意到名称缓存ID（cache ID）与RESULT CACHE操作相关联。如果你知道此缓存ID，可以查询v$result_cache_objects视图来显示关于缓存数据的信息。下面的查询显示缓存的结果集已经发布了（也就是说，可以使用了），还显示了结果缓存是何时创建的，构建

它花费了多长时间（以百分之一秒为单位），在其中存储了多少记录，以及它被引用了多少次：

```
SQL> SELECT status, creation_timestamp, build_time, row_count, scan_count
  2  FROM v$result_cache_objects
  3  WHERE cache_id = '089x05gkvfuxq7wqg06u9z0zkb';

STATUS    CREATION_TIMESTAMP BUILD_TIME ROW_COUNT SCAN_COUNT
--------- ------------------ ---------- --------- ----------
Published 2013-12-11 10:27           95        81          2
```

提供结果缓存信息的视图还有v$result_cache_dependency、v$result_cache_memory和v$result_cache_statistics。

从11.2版本开始，指定result_cache hint并非启用结果缓存唯一可用的方式。另一种技术是在表级别指定，通过将result_cache子句设置为force，引用该表的所有查询的所有结果集都必须被缓存（除非指定no_result_cache hint）。注意result_cache子句的默认模式是default。这个技术对于包含主要用于读取的数据的表尤其有用。事实上，多亏这种技术，才可以不必更改应用程序就利用结果缓存。你需要做的全部事情就是指定在表级别上启用结果缓存。

注意在一个单独的查询中引用多个表时，所有这些表都必须已经通过result_cache子句设置为force来启用结果缓存。下面的SQL语句，是来自rc_query_table.sql脚本的一段摘录，展示如何为本节示例中查询所使用的三张表启用结果缓存：

```
SQL> ALTER TABLE sales RESULT_CACHE (MODE FORCE);

SQL> ALTER TABLE customers RESULT_CACHE (MODE FORCE);

SQL> ALTER TABLE products RESULT_CACHE (MODE FORCE);
```

为保证结果集的一致性（也就是说，无论结果集是从结果缓存中直接提供，还是由数据库的内容计算得来，都应该是一样的），每当查询所引用的表中发生了某些变化，依赖它的缓存条目就失效了（很快就会讨论一个使用远程对象的例外）。即使没有真正的变化发生，也是这样。例如，即使是一条SELECT FOR UPDATE，马上跟随着一个COMMIT，也会导致依赖于SELECT的表的缓存条目失效。这意味着当一张表卷入到一个事务中时，依赖这张表的缓存条目就会失效，而不是在缓存条目依赖的数据发生变化时失效。换句话说，依赖性不是细粒度跟踪的：与修改的记录是否会影响缓存结果无关。

下面是控制服务器结果缓存的动态初始化参数。

- result_cache_mode指定在什么样的情况下使用结果缓存。既可以将它设置为manual（也就是默认设置），也可以将它设置为force。使用manual，仅当通过result_cache这个hint或result_cache子句启用它的情况下，结果缓存才会起作用。使用force，结果缓存会用于除指定了no_result_cache hint以外的所有查询。因为在大多数情形下你都是只想为有限数量的查询启用结果缓存，我建议你保持这个初始化参数的默认值即可。
- result_cache_max_size指定可以用于结果缓存的共享池内存总量（按字节）。如果将它设置为0，则该特性被完全禁用。其默认值，比0要大一些，是根据共享池的大小得来的。该内存分配是动态的，因此这个初始化参数指定的只是上限。可以通过类似下面这样的查询显示当前分配的内存：

```
SQL> SELECT name, sum(bytes)
  2  FROM v$sgastat
  3  WHERE name LIKE 'Result Cache%'
  4  GROUP BY rollup(name);

NAME                         SUM(BYTES)
------------------------- ----------
Result Cache                   145928
Result Cache: Bloom Fltr         2048
Result Cache: Cache Mgr           208
Result Cache: Memory Mgr          200
Result Cache: State Objs         2896
                               151280
```

- result_cache_max_result指定结果缓存中任何一个单独的条目可以使用的result_cache_max_size的总量（按百分比）。默认值是5。允许的值范围是0~100。
- result_cache_remote_expiration指定基于远程对象的结果缓存中条目的有效时间长度（按分钟计）。这样做是有必要的，因为基于远程对象的条目的失效操作并不是在远程对象发生变化时执行的。反而，条目会在达到初始化参数指定的有效时长时失效。默认值是0，也就意味着基于远程对象查询的缓存是被禁用的。

result_cache_max_size和result_cache_max_result初始化参数仅可以在系统级别上进行更改。此外，在12.1多租户环境下，它们仅可以在CDB级别上进行设置。另外两个参数，result_cache_mode和result_cache_remote_expiration，还可以在会话级别上进行修改。

警告 将result_cache_remote_expiration初始化参数设置为一个大于0的值会导致过期的结果。因此，只有在完全理解并接受这样做的影响的情况下，才应该使用大于0的值。

在结果缓存的使用方面，有几个尽管很明显但还是要提一下的限制。

- 引用具有不确定性的SQL函数、序列以及临时表的查询不会被缓存。
- 违反读一致性的查询不会被缓存。举例来说，在引用的表上拥有未决事务的情况下，由一个会话创建的结果集无法被缓存。
- 引用数据字典视图的查询不会被缓存。

DBMS_RESULT_CACHE

可以使用dbms_result_cache包来管理结果缓存。为此，它提供了以下子程序。

- bypass在会话或系统级别上临时禁用（或启用）结果缓存。
- flush从结果缓存中移除所有对象。
- invalidate使所有依赖于某个给定数据库对象的结果集无效。
- invalidate_object使一个单独的缓存条目无效。
- memory_report生成一份内存使用的报告。
- status显示结果缓存的状态。

2. PL/SQL函数结果缓存

PL/SQL函数结果缓存类似于服务器结果缓存，但是它支持PL/SQL函数。它也与服务器结果缓存一样共享相同的内存结构。它的用途是在结果缓存中存储PL/SQL函数返回的值（而且只有函数的返回值，结果缓存不能用于输出参数）。显然，拥有不同输入值的函数被缓存在各自的缓存条目中。下面的例子，是一段由rc_plsql.sql脚本生成的输出的摘录，展示了一个启用结果缓存的函数。要启用它，需要指定RESULT_CACHE子句：

```
SQL> CREATE OR REPLACE FUNCTION f(p IN NUMBER)
  2    RETURN NUMBER
  3    RESULT_CACHE
  4  IS
  5    l_ret NUMBER;
  6  BEGIN
  7    SELECT count(*) INTO l_ret
  8    FROM t
  9    WHERE id = p;
 10    RETURN l_ret;
 11  END;
 12  /
```

在下面的例子中，该函数在没有利用结果缓存（通过过程bypass，缓存被临时禁用了）的情况下被调用了10 000次。执行持续了大约4秒钟：

```
SQL> execute dbms_result_cache.bypass(bypass_mode => TRUE, session => TRUE)

SQL> SELECT count(f(1)) FROM t;

COUNT(F(1))
-----------
      10000

Elapsed: 00:00:04.02
```

现在，我们再次调用该函数10 000次，但是这一次是在结果缓存启用的情况下。执行只持续了大约百分之三秒：

```
SQL> execute dbms_result_cache.bypass(bypass_mode => FALSE, session => TRUE)

SQL> SELECT count(f(1)) FROM t;

COUNT(F(1))
-----------
      10000

Elapsed: 00:00:00.03
```

从11.2版本开始，数据库引擎会自动发现一个函数依赖了哪些表。基于此信息，一旦某个事务修改了结果缓存条目依赖的一张表的任何数据，则可以自动使该结果缓存条目失效。

警告 在11.1版本中，只有在函数级别指定了RELIES_ON子句，结果缓存条目才会失效。RELIES_ON子句的用途是指定函数返回值依赖哪些表。这个信息对于缓存条目的失效十分重要。如果没有指定它，或包含错误的信息，当函数依赖的对象发生修改的时候就不会有条目失效。因此，就会出现过期的结果。

PL/SQL函数结果缓存的使用方面有一些限制。该结果缓存无法用于以下这些函数。

- 带有OUT或IN OUT参数的函数
- 使用调用者的权限定义的函数（此限制从12.1版本开始不再存在）
- 管道化表函数
- 在匿名块中定义的函数
- 函数的IN参数或返回值是以下类型的：LOB、REF CURSOR、对象和记录

此外，注意未处理的异常不会存储在结果缓存中。也就是说，如果一个函数引发一个异常，并且该异常被传播至调用者，那么同一个函数的下一次调用还会被再次执行。

3. 客户端结果缓存

客户端结果缓存是一种客户端缓存，存储由某些特定应用程序执行查询所产生的结果集，这些应用程序使用的Oracle数据库驱动构建于OCI库之上（例如JDBC OCI、ODP.NET、OCCI和ODBC）。它的用途和工作方式类似于服务器结果缓存。特别是，启用它们的可选技术（result_cache hint、RESULT_CACHE子句以及result_cache_mode初始化参数）是相同的。唯一额外的需求是，利用客户端语句缓存（client-side statement caching，参见第12章）的专有SQL语句可以启用客户端结果缓存。与服务器端实现相比，有两点重要的区别。第一，它避免了需要在客户端/服务器之间往返执行SQL语句。这是一个很大的优势。第二，失效是基于一种轮询机制，因此，一致性无法得到保证。这又是一个很大的劣势。

注意 在12.1版本的多租户环境下，客户端结果缓存不受支持。

为实现轮询，客户端不得不定期执行数据库调用，以向数据库引擎检查它的其中某个缓存结果集是否已经失效。为最小化与轮询相关的负载，每次当客户端因为其他什么原因执行数据库调用时，它也会同时检查缓存结果集的有效性。通过这种方式，对于那些有规律地执行“定期”数据库调用的客户端来说，就避免了专门用于检查缓存结果集有效性的数据库调用。

即使这是一种客户端缓存，你也不得不在服务器端启用它。注意即使服务器端缓存被禁用了（换句话说，如果将result_cache_max_size初始化参数设置为0），客户端缓存也会继续工作。以下是控制客户端结果缓存的初始化参数。

- client_result_cache_size指定每个客户端进程可以用于结果缓存的最大内存总量（按字节计）。如果将它设置为0，也就是默认值，则会禁用该特性。这个初始化参数是静态参数，并且仅可以在系统级别上进行设置。因此，要修改它，必须重启数据库实例。
- client_result_cache_lag指定两次数据库调用之间的最长时间间隔（按毫秒计）。也就是说，

它指定过期的结果集可以在客户端缓存中保留多长时间。默认值是3000。这个初始化参数是静态参数，并且仅可以在系统级别上进行设置。因此，要修改它，必须重启数据库实例。

除了服务器端配置之外，还可以在客户端的sqlnet.ora文件中指定以下参数。

- oci_result_cache_max_size覆盖使用client_result_cache_size初始化参数指定的服务器端配置。注意，不管怎样，如果客户端结果缓存在服务器端被禁用了，则这个参数也无能为力。
- oci_result_cache_max_rset_size指定任何一个单独的结果集可以使用的最大内存总量（按字节计）。
- oci_result_cache_max_rset_rows指定任何一个单独的结果集可以存储的最大行数。

15.2.2　何时使用

如果正在处理一个因为应用程序不断重复执行相同的操作而引起的性能问题，你必须减少执行的频率或者减少操作的响应时间。理想情况下，两者都应该做。然而，有时（例如，当应用程序的代码无法修改）你只能实现后者。为减少响应时间，应该首先尝试使用第13章和第14章中呈现的技术。如果这样还不够，只有那时候才应该考虑高级优化技术，例如结果缓存。基本上来说，结果缓存在两个给定条件下是高效的。第一，相同的数据查询远比修改要频繁。第二，有足够的内存来缓存结果集。

在大多数情况中，你不应该为所有的查询启用结果缓存。事实上，大多数时候，只有特定的查询能够从结果缓存中获益。对于除了特殊案例以外的其他查询，结果缓存管理就只是纯粹的负载，并有可能使缓存压力过大。还要牢记服务器端缓存由所有会话共享，所以它们的访问是同步的（它们可以像其他任何共享资源那样变成序列化的）。因此，你应该只为需要结果缓存的查询、子查询以及表来启用该项技术。换句话说，只有在当结果缓存对于改进性能是真正有必要的情况下，才应该有选择性地启用它。

服务器结果缓存不会完全避免执行一个查询的负载。这意味着如果一个查询在没有使用结果缓存的情况下已经实现了相对较少的逻辑读（并且没有物理读），那么使用结果缓存的时候也不会快到哪去。记住，缓冲区缓存和结果缓存都存储在相同的共享内存中。

PL/SQL函数结果缓存对于经常从SQL语句中调用的函数尤其有用。事实上，对于这样的函数来讲，即使输入参数只在一少部分记录上有所不同，对每一行处理的或返回的数据都执行调用也很常见。不管怎样，经常在PL/SQL中被调用的函数也可以从结果缓存中获益。

因为一致性的问题，应该仅将客户端结果缓存用于只读表或只读为主的表。

最后，记住你可以同时利用服务器端和客户端结果缓存。然而，对于由客户端执行的查询，你不能选择绕过客户端结果缓存而只使用服务器结果缓存。也就是说，两种结果缓存都会被使用。

15.2.3　陷阱和谬误

正如在之前的章节中指出的那样，结果的一致性在以下情况下是无法确保的：

- 将result_cache_remote_expiration初始化参数设置为一个大于0的值，并且通过数据库链接（dblink）执行查询时；
- 在11.1版本中，定义了没有指定（或错误指定）RELIES_ON子句的PL/SQL函数时；
- 使用了客户端结果缓存时。

因此，遇到这样的情况，最好避免使用结果缓存，除非你完全理解并接受以上每种情况可能带来的影响。

对于引用了非确定性PL/SQL函数的查询，或者对于类似NLS参数和上下文环境这样的会话级设置敏感的函数来说，如果缓存了它们的结果，则它们可能不会像你预期的那样工作。问题是在默认情况下，数据库引擎认为那些函数是确定性的，所以最后可能就会生成错误的结果。我在rc_query_nondet.sql脚本中提供了几个例子。下面的例子就是其中之一（注意第二个查询是如何返回错误结果的）：

```
SQL> CREATE OR REPLACE FUNCTION f RETURN VARCHAR2
  2  IS
  3    l_ret VARCHAR2(64);
  4  BEGIN
  5    SELECT /*+ no_result_cache */ to_char(sysdate) INTO l_ret FROM dual;
  6    RETURN l_ret;
  7  END f;
  8  /

SQL> ALTER SESSION SET nls_date_format = 'YYYY-MM-DD HH24:MI:SS';

SQL> SELECT /*+ result_cache */ f() FROM dual;

F()
-------------------
2014-01-06 18:08:05

SQL> ALTER SESSION SET nls_date_format = 'YYYY-MM-DD';

SQL> SELECT /*+ result_cache */ f() FROM dual;

F()
-------------------
2014-01-06 18:08:05
```

为避免这个问题，从11.2.0.4版本开始，可以将未公开初始化参数_result_cache_deterministic_plsql设置为TRUE。有关详细信息，请参考Oracle Support文档*Bug 14320218 Wrong results with query results cache using PL/SQL function*（14320218.8）。

15.3 并行处理

向数据库引擎提交一条SQL语句时，默认情况下它是由一个单独的服务进程串行执行的。因此，即使运行数据库引擎的服务器有多个CPU内核，你的SQL语句也只能运行在其中一个CPU内核上。并行处理的作用是将一条单独SQL语句的执行过程分布在多个CPU内核上。

注意 并行处理仅在企业版中可用。

15.3.1 工作原理

在描述如何通过并行方式执行查询、DML语句和DDL语句的具体细节之前，有必要先理解并行处理的基础，如何配置一个数据库实例以便利用并行处理，以及如何控制并行度。

1. 基础

不使用并行处理时，一条SQL语句是由一个单独的服务进程串行执行的，也就是在一个单独的CPU内核上运行。这意味着能够用于执行一条SQL语句的资源总量受到单一CPU内核能够提供的处理能力限制。举例来说，如图15-5所示，如果SQL语句执行的扫描整个段的数据访问操作（如果大部分数据是从磁盘读取，则可能是磁盘I/O密集型操作）与磁盘I/O子系统能够传递的总吞吐量无关，则响应时间会受到一个单独的CPU内核所能够使用的带宽限制。因为CPU内核与磁盘之间的数据访问路径的硬件限制，带宽一定是受限的，而且当执行是串行的时候还无法完全利用这部分带宽：当服务进程获取CPU，按照定义我们知道它此时没有访问磁盘（异步I/O操作对于这里来说是一个例外），因此无法利用磁盘I/O子系统能够传递的全部吞吐量。

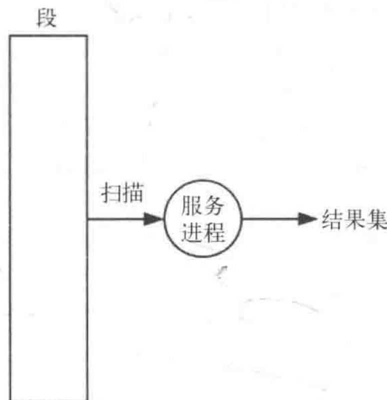


图15-5 串行执行的SQL语句由单独的一个服务进程处理

下面的SQL语句及其关联的执行计划展示了图15-5所示处理过程的一个例子：

```
SELECT * FROM t

----------------------------------
| Id  | Operation         | Name |
----------------------------------
|   0 | SELECT STATEMENT  |      |
|   1 |  TABLE ACCESS FULL| T    |
----------------------------------
```

并行处理的目的是将一个大的任务拆分成几个小的子任务。如果有一条并行处理的SQL语句存在，基本上意味着有多个并行查询从属进程（parallel query slave process），为简单起见，本书全部称作从属进程（slave process）合作执行单独的一条SQL语句。与提交该SQL语句的会话关联的服务进程控制从属进程之间的协调。因为这一角色，经常将该服务进程称作查询协调器（query coordinator）。查询协调器负责获得从属进程，为每一个进程分配一项子任务，收集并组合它们传递过来的部分结果

集，并将最终的结果集返回给客户端。例如，在SQL语句为整个段执行扫描的一个例子中，查询协调器能够指导每个从属进程去扫描该段的一部分并向它传递必要的数据。图15-6演示了这个过程。因为这四个从属进程中的每一个都能够在一个不同的CPU内核上运行，在此情况下响应时间不再受到单一CPU内核能够使用的带宽的限制。

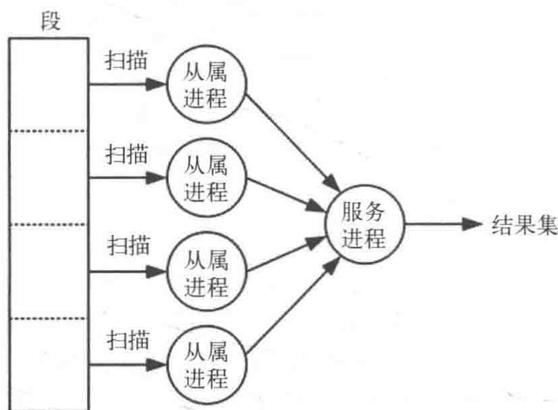


图15-6 并行执行的SQL语句由一个称作查询协调器的服务进程协调的多个从属进程所处理

在一次类似图15-6所示的并行扫描中，工作是在从属进程中间按照称作粒度单元（granule）的单位进行分配。每个从属进程，在任意给定的时间内，只处理一个单独的粒度单元。如果粒度单元的数量多于从属进程的数量，当一个从属进程处理完毕一个粒度单元之后，它会接收到另一个要处理的粒度单元，直到所有的粒度单元都被处理完毕。数据库引擎可以使用以下两种类型的粒度单元。

- 分区粒度（partition granule）是一整个分区或子分区。显然，这种类型的粒度单元只能用于已分区段。
- 块范围粒度（block range granule）是来自一个段在运行时（不是在解析时）动态定义的一系列块。

注意 对于一张外部表的并行扫描，会将粒度单元定义为一个外部文件的一部分（默认大小是10 MB）。所以没有必要使用多个外部文件来允许并行访问。

因为分区粒度的定义是静态的（只有数量会因为分区裁剪而发生变化），大多数时候更倾向于使用块范围粒度。块范围粒度的主要优势是，在大多数情况下，它们具备将工作向从属进程均匀分布的条件。事实上，使用分区粒度，工作的分布不仅强烈依赖分区数量和从属进程数量之间的比值，而且还依赖每个分区上存储的数据总量。假设每个分区包含近似相等的数据总量。在这种情况下，要对工作有一个合理的分配，分区的数量应该是从属进程数量的几倍。如果没有将工作均匀分布，某些从属进程可能比其他的多做许多工作，因此就会导致更长的响应时间。结果就是，并行执行的综合效率可能会受到损害。

下面的执行计划展示了图15-6所示处理过程的一个例子：

```
SELECT * FROM t

-----------------------------------------------------------------------------
| Id  | Operation              | Name     |    TQ  |IN-OUT| PQ Distrib |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |          |        |      |            |
|   1 |  PX COORDINATOR        |          |        |      |            |
|   2 |   PX SEND QC (RANDOM)| :TQ10000 |  Q1,00 | P->S | QC (RAND)  |
|   3 |    PX BLOCK ITERATOR |          |  Q1,00 | PCWC |            |
|   4 |     TABLE ACCESS FULL| T        |  Q1,00 | PCWP |            |
-----------------------------------------------------------------------------
```

该执行计划按以下方式执行。

(1) 通过操作4（TABLE ACCESS FULL），每个从属进程扫描表的一部分。从属进程具体扫描哪个部分取决于它的父操作3（PX BLOCK ITERATOR）。这是与块范围粒度有关的操作。

(2) 操作2（PX SEND QC）将检索到的数据发送给查询协调器。

(3) 查询协调器通过操作1（PX COORDINATOR）从从属进程接收数据并将其发送回客户端。

当进程之间发生通信的时候，发送数据的进程称作*生产者*（producer），而接收数据的进程则称作*消费者*（consumer）。为发送数据，生产者向一个称作*表队列*的队列写入数据。为接收数据，消费者从表队列读取数据。根据由生产者和消费者执行的操作，数据使用以下方法之一进行分发（在dbms_xplan包的输出中，PQ Distrib列提供了此信息）。

- 广播（Broadcast）：每个生产者向每个消费者发送所有数据。
- 本地广播（Broadcast Local）：这是广播分布的一种变形。它用于仅向一部分的从属进程发送所有数据。它通常在RAC环境中对于最小化跨实例通信比较有帮助。
- 循环（Round-robin）：生产者每次只向一个消费者发送一行数据，就像发纸牌。因此，数据在消费者之间均匀分布。
- 范围（Range）：生产者向不同的消费者发送特定范围的数据。会执行动态范围分区以确定应该将哪一行数据发送给哪一个消费者。例如，对于一个排序，这个方法基于ORDER BY子句中使用的列对数据进行范围分区，所以每个消费者可以只排序它自己那部分数据。
- 散列（Hash）：生产者根据散列函数向消费者发送数据。此过程中会执行动态散列分区以确定应该将哪一行数据发送给哪一个消费者。例如，对于一个聚合，这种方法可能会根据GROUP BY子句中使用的列对数据进行散列分区。
- 分区键（Partition Key）：生产者根据分区键向消费者发送数据。有关更多信息，请参考14.7.2节。
- 混合散列（Hybrid Hash）：生产者或者使用广播或者使用散列分布方法向消费者发送数据。使用这两者中的哪一个在运行时决定。仅从12.1版本开始可用。
- 单一从属进程（One Slave）：生产者向一个单独的消费者发送所有数据。仅从12.1版本开始可用。
- 查询协调器随机（QC Random）：每个生产者都将所有数据发送给查询协调器。顺序并不重要（因此是随机的）。这是最常见的与查询协调器通信所使用的分布方式。
- 查询协调器排序（QC Order）：每个生产者都将所有数据发送给查询协调器。此时顺序很重要。例如，并行执行的排序会使用这种方法将数据发送给查询协调器。

并行操作之间的关系

在并行执行的执行计划中会用到的并行操作之间的关系如下所示。

- 并行至串行（P->S）：一个并行操作向一个串行操作发送数据。例如，这个操作用于在执行计划中向查询协调器发送数据。
- 并行至并行（P->P）：一个并行操作向另一个并行操作发送数据。有两组从属进程存在的时候会用到这个操作。
- 与父操作组合的并行操作（PCWP）：一个被并行执行的操作，相同的从属进程还会执行该执行计划中的父操作。因此，不会有通信发生。
- 与子操作组合的并行操作（PCWC）：一个被并行执行的操作，相同的从属进程还会执行该执行计划中的子操作。因此，不会有通信发生。
- 串行至并行（S->P）：一个串行执行的操作向一个并行执行的操作发送数据。因为大多数时候这是没有效率的，它应该被避免。没有效率的一个主要的原因：一个单独的进程能够产生数据的速度，可能无法跟上多个进程能够消耗的速度。如果就是这种情况，消费者就会花费大量的时间等待数据而不是做真正的在工作。
- 与父操作组合的串行操作（SCWP）：一个被串行执行的操作，相同的从属进程还会执行该执行计划中的父操作。因此，不会有通信发生。从12.1版本开始可用。
- 与子操作组合的串行操作（SCWC）：一个被串行执行的操作，相同的从属进程还会执行该执行计划中的子操作。因此，不会有通信发生。从12.1版本开始可用。

在由dbms_xplan包生成的输出中，并行操作之间的关系在IN-OUT列中提供。

以一个序列形式执行的多个并行操作集体称为*数据流操作*（data flow operation，DFO）。在很多情况下，执行计划拥有一个单独的数据流操作。但是，也存在需要多个数据流操作的情况。要想知道数据流操作的数量，必须检查由dbms_xplan包生成的输出中的TQ列。通过它，不仅可以确定在一个执行计划中使用了多少个数据流操作，而且还可以知道哪些操作是由哪一组从属进程执行的。事实上，TQ列的内容提供以下信息。

- 值为NULL与查询协调器执行的操作对应。
- 前缀为字母Q的值与数据流操作的ID对应。
- 跟在逗号后面的值与一组从属进程写入数据的表队列的ID对应。你无法知晓有多少个从属进程属于这一组。

在前面的执行计划中，因为值Q1,00的缘故，你知道操作2到操作4属于一个单独的数据流操作（ID是1的那些），而且它们都只是被单独一组从属进程执行的（ID是0的那些）。

数据访问操作并非唯一可以并行执行的操作。事实上，数据库引擎也能够并行化插入、联接、聚合和排序等操作。当一条SQL语句执行两个或更多的独立操作（例如，扫描和排序）时，数据库引擎通常会使用两组从属进程。举例来说，如图15-7所示，如果一条SQL语句执行一次扫描，然后是一次排序，则一组用于扫描而另外一组用于排序。

一个独立操作的并行化称为*内部操作并行*（intra-operation parallelism）。举例来说，在图15-7中，

内部操作并行（使用四个从属进程）被使用了两次：一次用于扫描，一次用于排序。当两组从属进程被用于执行一个数据流操作，该并行化称为交互操作并行化（inter-operation parallelism）。举例来说，在图15-7中，交互操作并行化被用于第一组进程（扫描）和第二组进程（排序）之间。

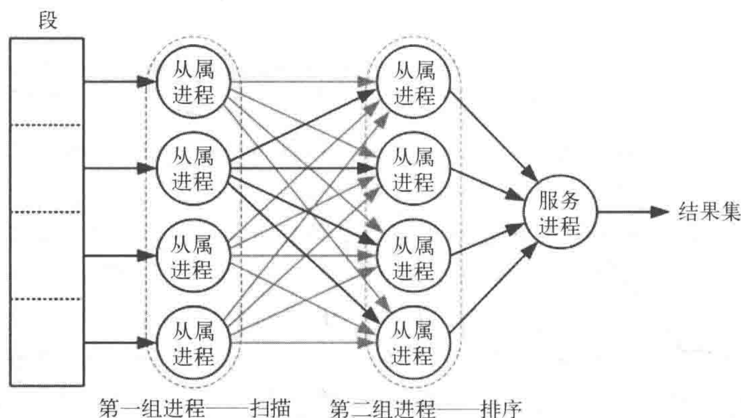


图15-7　两组从属进程可以用于执行同一条SQL语句

下面的执行计划是图15-7所示处理过程的一个例子：

```
SELECT * FROM t ORDER BY id

---------------------------------------------------------------------------
| Id  | Operation               | Name      |    TQ  |IN-OUT| PQ Distrib |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |           |        |      |            |
|   1 |  PX COORDINATOR         |           |        |      |            |
|   2 |   PX SEND QC (ORDER)    | :TQ10001  |  Q1,01 | P->S | QC (ORDER) |
|   3 |    SORT ORDER BY        |           |  Q1,01 | PCWP |            |
|   4 |     PX RECEIVE          |           |  Q1,01 | PCWP |            |
|   5 |      PX SEND RANGE      | :TQ10000  |  Q1,00 | P->P | RANGE      |
|   6 |       PX BLOCK ITERATOR |           |  Q1,00 | PCWC |            |
|   7 |        TABLE ACCESS FULL| T         |  Q1,00 | PCWP |            |
---------------------------------------------------------------------------
```

上面的执行计划是由一个单独的数据流操作组成（ID为1那些）。操作5到7在列TQ上拥有相同的值（Q1,00），也就是说它们是被同一组从属进程执行的（图15-7中的第一组）。另一方面，操作2到操作4拥有另一个值（Q1,01），因此它们是被另一组从属进程执行的（图15-7中的第二组）。第一组是生产者，基于块范围粒度扫描表（操作6）并将检索到的数据发送给第二组。接下来，第二组作为消费者，接收数据，对其排序，然后将排序的结果集发送给查询协调器。第一组和第二组同步进行它们的处理过程。因为这两组从属进程相互通信，处理数据较快的那一组会等待另外一组。

2. 基本配置

本部分主要描述要成功设置一个用于并行处理的数据库实例，你必须要了解的基本初始化参数。这些初始化参数涉及从属进程池和内存利用。

● **从属进程池**

每个数据库实例的最大从属进程数量是有限的，并且由一个数据库实例以从属进程池的方式进行维护。查询协调器从池中请求从属进程，使用它们来执行一条SQL语句，最终，当执行完成时，将它们返还给这个池。可以通过设置下面的初始化参数来配置这个池。

- parallel_min_servers指定在数据库实例启动时启动的从属进程数量。这些从属进程总是处于可用状态，而且当一个服务进程请求它们的时候不需要启动。当请求超过这个最小数量时，会以动态方式启动从属进程，而且一旦返还给进程池，会保持空闲状态5分钟。如果在这一时间段内没有被重用，就会关闭它们。默认情况下，会将这个初始化参数设置为0[①]。这意味着在最初的时候不会启动任何从属进程。我建议只在当一些SQL语句花费过长的时间等待从属进程的启动时才去更改这个值。与这个操作关联的等待事件是os thread startup。
- parallel_max_servers指定池中可用从属进程的最大数量。很难给出关于如何设置这个参数的建议。虽然如此，CPU内核数量的10~20倍是一个不错的开始。默认值还依赖于其他几个初始化参数、版本以及平台。可以将parallel_max_servers的最大值设置为比processes初始化参数的值小15的数值。如果尝试将parallel_max_servers设置为一个更高的值，那么在数据库实例启动的时候，会自动调整parallel_max_servers，并且会在alert日志中写入一条消息。

要显示池的状态，可以使用下面的查询：

```
SQL> SELECT *
  2  FROM v$px_process_sysstat
  3  WHERE statistic LIKE 'Servers%';

STATISTIC          VALUE
------------------ -----
Servers In Use         4
Servers Available      8
Servers Started       46
Servers Shutdown      34
Servers Highwater     12
Servers Cleaned Up     0
```

在RAC环境中，每个数据库实例都是群集的一部分，且拥有它们自己的从属进程池。当以并行方式执行一条SQL语句时，从属进程既可以从本地分配，也可以从一个单独的实例远程分配，或从多个数据库实例分配。下面的方法可以用于控制从哪些数据库实例分配从属进程。

- parallel_instance_group和instance_groups初始化参数可以用于将从属进程的分配限制到指定的数据库实例上。通过instance_groups初始化参数，你指定每个数据库实例所处的实例组。通过parallel_instance_group初始化参数，你指定从属进程从哪个实例组分配。从11.1版本开始，不再赞成使用instance_groups初始化参数。所以刚刚描述的这种方法仅适用于10.2版本。
- 从11.1版本开始，从属进程的分配是服务感知的。从属进程仅从特定实例中分配，这些实例与通过并行方式执行SQL语句的会话所在实例处于相同的服务之下。从11.1版本开始，这种方法取代了前一种。
- 从11.2版本开始，将parallel_force_local初始化参数设置为TRUE（默认值是FALSE）时只会分配本地的从属进程。

① 自12.1版本起，parallel_min_servers初始化参数为"cpu_count*parallel_threads_per_cpu*2"，而不是此处的0。

因为从属进程池的配置是具体到数据库实例的，在12.1多租户环境下，不能在PDB级别设置parallel_min_servers和parallel_max_servers初始化参数。但是，可以通过数据库服务或parallel_force_local初始化参数在PDB级别控制从属进程的分配。

- **内存利用**

用于在进程之间通信的表队列是可以从shared pool或large pool中分配的内存结构。然而，不推荐为它们使用shared pool。Large pool专用于不可重用的内存结构，是一个更好的选择。以下两个配置可以引导表队列使用large pool。

- 自动SGA管理是通过sga_target或memory_target启用的（后者仅从11.1版本开始可用）。
- 将parallel_automatic_tuning初始化参数设置为TRUE。注意，这个初始化参数是不再赞成使用的。然而，如果不想使用自动SGA管理，设置它是引导并行处理使用large pool的唯一途径。

注意　不去管它的名称，parallel_automatic_tuning初始化参数只做两件简单的事情。第一，它改变几个与并行处理有关的初始化参数的默认值。第二，它通知数据库引擎为表队列使用large pool。

每个表队列是由三个缓冲区（使用RAC时最多五个）组成，这些缓冲区是与通过表队列通信的每一对进程相对应的。每个缓存的大小（按字节计）是通过parallel_execution_message_size初始化参数设置的。其默认值取决于数据库引擎版本。直到11.1版本，它要么是2152个字节，要么是将parallel_automatic_tuning初始化参数设置为TRUE时的4096个字节。从11.2版本开始，默认大小是16 KB。出于最佳性能考虑，应该将它设置为能够支持的最大值。取决于所使用的平台，可以是16 KB、32 KB或64 KB。因此，尤其是在11.2版本之前，我建议你修改其默认值。

当增大parallel_execution_message_size初始化参数的值时，应该确保有足够的内存可用。可以使用公式15-1来估算对于非RAC数据库实例large pool中应该保持可用的最大内存总量。出于这个目的，这个公式计算在需要两组从属进程并且使用可能的最大并行度（parallel_max_servers初始化参数的一半）执行时，必要的缓存数量是多少。注意，在RAC环境中，不仅每一对进程之间通信的缓存数量可以更高一些（最高到5而非3），而且最大并行度也取决于实例的数量。

公式15-1　表队列的非RAC数据库实例使用的large pool内存总量

$$large_pool_size > 3 \cdot \left(parallel_max_servers + \frac{parallel_max_servers^2}{4} \right) \cdot parallel_execution_message_size$$

要显示数据库实例目前正在使用的large pool内存有多少，可以运行下面的查询：

```
SQL> SELECT *
  2  FROM v$sgastat
  3  WHERE name = 'PX msg pool';

POOL       NAME              BYTES
---------- ----------- ---------
large pool PX msg pool 823296000
```

可以运行下面的查询来显示当前分配的表队列缓存数量（Buffers Current统计信息），以及自数据库实例启动以来一次分配过的最大数量（Buffers HWM统计信息）：

```
SQL> SELECT *
  2  FROM v$px_process_sysstat
  3  WHERE statistic IN ('Buffers Current                ',
  4                      'Buffers HWM                    ');

STATISTIC                      VALUE
------------------------------ -----
Buffers Current                45076
Buffers HWM                    49924
```

因为内存配置是具体到数据库实例的，所以在12.1多租户环境下，不可能在PDB级别设置parallel_automatic_tuning和parallel_execution_message_size初始化参数。

在RAC环境中每个数据库实例可以拥有它自己的内存设置。唯一的例外是parallel_execution_message_size初始化参数。事实上，如果将这个参数设置为不同的值，则数据库实例之间无法互相通信。这种情况下，执行一条涉及多个数据库实例的SQL语句时，就会引发一个错误。下面是此类错误的一个例子：

```
SQL> SELECT * FROM gv$instance;
SELECT * FROM gv$instance
       *
ERROR at line 1:
ORA-12850: Could not allocate slaves on all specified instances: 2 needed, 1 allocated
ORA-12801: error signaled in parallel query server P001, instance 32766
```

3. 并行度

用于内部操作并行化的从属进程数量称作*并行度*（DOP）。因为并行度规定用于内部操作并行化的从属进程数量，所以在使用交互操作并行化时，用于执行一条SQL语句的从属进程数量要比并行度高一些。无论如何，一个单独的数据流操作无法使用高于并行度两倍数量的从属进程。举例来说，图15-7展示了一个从属进程的数量是并行度两倍的例子。

当并行处理一条SQL语句（或者其中一部分）的时候，数据库引擎不得不选择用于此目的的并行度。尽管有几个初始化参数和其他的因素决定实际的并行度，但实际上只有两种主要的模式可以用于设置一个数据库实例以控制并行度。

- *手动并行度*：在这种模式下，可以在会话、对象或SQL语句中的任意一个级别控制并行度。
- *自动并行度*：在这种模式下，数据库引擎自动为每条SQL语句选择最优的并行度。

手动并行度控制是11.1及之前版本中唯一可用的模式。

警告　建议仅从11.2.0.3版本开始使用自动并行度。原因是在此之前的版本中，有几个bug使得自动控制很难实现。请参考Oracle Support文档 *Init.ora Parameter "PARALLEL_DEGREE_POLICY" Reference Note*（1216277.1）获取更多信息。还要注意，在11.2.0.2版本中，自动并行度的功能性方面引入了一些变化。我不觉得在11.2.0.3之前的版本中使用它有多大意义，所以11.2.0.1版本的功能在本书中不做介绍。

从11.2版本开始，parallel_degree_policy初始化参数不仅用于在手动和自动并行度之间进行选择，而且还会启用其他与并行处理相关的特性。它接受以下值。

- manual：启用手动并行度。这是默认值。
- limited：只为引用了将PARALLEL设置为DEFAULT（详细信息见下）的对象的SQL语句启用自动并行度。对于其他语句，使用手动并行度。
- auto：为所有SQL语句启用自动并行度。此外，还启用了其他两个特性：并行语句排队（parallel statement queuing）和内存中并行执行（in-memory parallel execution）。
- adaptive：这种模式类似于auto。唯一的区别是同时还启用了性能反馈（performance feedback）。这个值仅从12.1版本开始可用。

注意 并行语句排队、内存中并行执行，以及性能反馈都与自动并行度无关。因此令人遗憾的是，它们都是被同一个单独的初始化参数激活的。如果可以有选择性地激活它们就好多了。

---

parallel_degree_policy初始化参数可以在会话级别进行设置，从12.1版本开始可以在PDB级别进行设置。也可以在SQL语句级别通过使用语句级别的语法指定parallel hint来覆盖它的值。parallel hint支持以下值。

- parallel(manual) 激活手动并行度。
- parallel(auto) 激活自动并行度（但是不会激活并行语句排队和内存中并行执行）。
- parallel(*n*) 将并行度设置为作为参数(*n*)指定的整数值。

警告 从11.2版本开始，parallel hint支持两种语法：语句级别和对象级别。语句级别的语法，就像刚才描述的，在SQL语句级别覆盖parallel_degree_policy初始化参数。对象级别的语法，在接下来的“手动并行度”一节中讲述，会覆盖与表以及索引相关的并行度。

下面几个部分的主要目标，是描述在不考虑从属进程数量可能会被系统的负载或其他因素限制的情况下，数据库引擎如何决定并行度。稍后，“限制并行度”部分会描述并行度可能会被减少的情况。

● 默认并行度

通常所谓的默认并行度，实际上可能是你想为任何一个并行SQL语句使用的最大并行度。事实上，只有在任意给定时间点最多有运行一条SQL语句时默认值才运行良好。如何使用以及何时使用默认值取决于多个因素，比如数据库实例配置。接下来的两个小节，在讨论手动和自动并行度是如何工作的时候，会提供更多关于默认并行度的信息。

为计算默认并行度，如公式15-2所示，数据库引擎将cpu_count初始化参数的值乘以一个CPU内核预期可以处理的从属进程数量（parallel_threads_per_cpu初始化参数）。延伸到RAC环境中时，结果值要进一步乘以群集中数据库实例的数量。

公式15-2 默认并行度是你可能想为任何一条并行SQL语句使用的最大并行度

$$default_dop = cpu_count \cdot parallel_threads_per_cpu \cdot number_of_instances$$

提示 在大多数平台上，parallel_threads_per_cpu初始化参数的默认值是2。假如在CPU级别启用了

多线程，结果就是，cpu_count初始化参数的值被人为地提高了，我建议此时将parallel_threads_per_cpu初始化参数设置为1。

- **手动并行度**

每张表和每个索引都有一个关联的并行度。对于引用该对象的操作默认会使用这个并行度。它的默认值是1，也就是说不使用并行处理。如下面SQL语句所示，并行度通过使用PARALLEL子句设置，既可以在创建对象时使用，也可以在稍后使用：

```
CREATE TABLE t (id NUMBER, pad VARCHAR2(1000)) PARALLEL 4

ALTER TABLE t PARALLEL 2

CREATE INDEX i ON t (id) PARALLEL 4

ALTER INDEX i PARALLEL 2
```

警告 使用并行处理来改进维护任务或创建表和索引的批处理任务等操作性能的做法十分常见。出于这个目的，通常会指定PARALLEL子句。但是要知道，使用这个子句时，该并行度不仅用于表或索引的创建期间，而且日后引用这些表或索引的操作中也会使用该并行度。因此，如果只想在表或索引的创建期间使用并行处理，一定要记得在创建完毕后修改并行度。

要禁用并行处理，要么将并行度设置为1，要么指定NOPARALLEL子句：

```
ALTER TABLE t PARALLEL 1

ALTER INDEX i NOPARALLEL
```

在没有指定并行度的情况下使用PARALLEL子句时（例如，ALTER TABLE t PARALLEL），应使用默认并行度。因为默认值只是在你想在任意给定时间内执行至多一条SQL语句时有好处，我通常推荐还是指定一个值。

要覆盖在表或索引级别上定义的并行度，可以使用parallel、no_parallel、parallel_index以及no_parallel_index这几个hint。事实上，当这些hint和对象级别语法一起使用时，前两个覆盖表级别的设置，第三个和第四个覆盖索引级别的设置。我强调一下对象级别的语法，指定对象名称或别名的那个语法，必须使用。下面是使用这个语法时不仅指定对象名称（t是指表，i是索引）而且还有并行度（16）的例子：

```
SELECT /*+ parallel(t 16) */ * FROM t

SELECT /*+ parallel_index(t i 16) */ * FROM t
```

通过parallel这个hint，还可以显式要求使用默认并行度：

```
SELECT /*+ parallel(t default) */ * FROM t
```

在一个单独的数据流操作中为不同的表或索引指定不同的并行度时，数据库引擎会为整个数据流操作计算一个单独的并行度。一般而言，所选择的并行度就是在表或索引级别指定的最大的那个。

- **自动并行度**

使用自动并行度的想法非常简单：对于每一条SQL语句，由查询优化器选择最优的并行度。因此，查询优化器根据执行计划和预期需要完成的处理总量调整并行度。作为一个例子，图15-8展示了当我在不同大小的表上执行全表扫描的时候，查询优化器在我的测试服务器上选择的并行度。注意为执行这个测试，我执行了px_dop_auto.sql脚本。

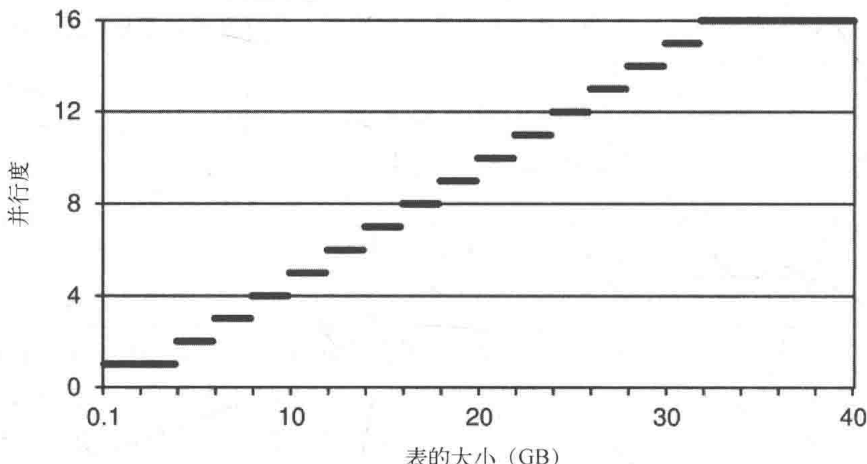


图15-8 在最小值（1）和最大值（16）之间，并行度随着处理总量（发生全表扫描的段大小）成比例增加

图15-8显示，一方面，在一个阈值之下查询优化器会决定以串行方式运行一条SQL语句，而另一方面，也存在一个不会被逾越的最大并行度。

该阈值的定义是，在考虑并行处理之前，一条SQL语句以串行方式执行应该持续的最小时间总量（根据查询优化器的估算）。它是通过parallel_min_time_threshold初始化参数配置的。默认值是自动的，当前是等于10秒钟。如果想要更改那个阈值，可以设置parallel_min_time_threshold初始化参数来满足你预期的秒数。

最大并行度取决于parallel_degree_limit初始化参数。可以将它设置为以下值之一。

- CPU：最大并行度等于默认并行度。此为默认值。
- IO：最大并行度由磁盘I/O上限定义。参考本章稍后的“磁盘I/O上限”部分以获取关于它的额外信息。
- 一个整数值显式指定最大并行度。

要使用自动并行度，必须满足两个条件。首先，通过I/O口径收集的统计信息必须可用。这些统计信息都是什么以及如何收集它们会在下一部分“磁盘I/O上限”中介绍。其次，该特性必须通过parallel_degree_policy初始化参数或parallel(auto) hint来启用。将parallel_degree_policy初始化参数设置为limited时，有一个额外的要求：只会对那些拥有相关的默认并行度的表和索引考虑使用并行处理。下面的例子，来自px_dop_limited.sql脚本的输出，证实了这一点：

```
SQL> ALTER SESSION SET parallel_degree_policy = limited;
```

```
SQL> ALTER TABLE t NOPARALLEL;

SQL> SELECT * FROM t;

---------------------------------
| Id | Operation         | Name |
---------------------------------
|  0 | SELECT STATEMENT  |      |
|  1 |  TABLE ACCESS FULL| T    |
---------------------------------

SQL> ALTER TABLE t PARALLEL;

SQL> SELECT * FROM t;

-----------------------------------------------------------------------
| Id | Operation             | Name     |    TQ  |IN-OUT| PQ Distrib |
-----------------------------------------------------------------------
|  0 | SELECT STATEMENT      |          |        |      |            |
|  1 |  PX COORDINATOR       |          |        |      |            |
|  2 |   PX SEND QC (RANDOM)| :TQ10000 |  Q1,00 | P->S | QC (RAND)  |
|  3 |    PX BLOCK ITERATOR |          |  Q1,00 | PCWC |            |
|  4 |     TABLE ACCESS FULL| T        |  Q1,00 | PCWP |            |
-----------------------------------------------------------------------

Note
-----
   - automatic DOP: Computed Degree of Parallelism is 4 because of degree limit
```

上面例子中由dbms_xplan包生成的Note部分最后一行输出清晰地表明是否使用了自动并行度。而且，如果使用了，还会提到选择的并行度大小。其他可以出现在Note部分与自动并行度相关的消息如下：

```
automatic DOP: Computed Degree of Parallelism is 2

automatic DOP: Computed Degree of Parallelism is 1 because of parallel threshold

automatic DOP: skipped because of IO calibrate statistics are missing
```

如果由查询优化器选择的并行度不合理，从12.1版本开始，可以通过parallel_degree_level初始化参数调整其估算。它的默认值是100。如果你指定的值低于100，并行度会按比例减小。举例来说，使用值50时，并行度减少50%，如果你指定的值高于100，并行度会按比例增加。举例来说，使用值200时，假如没有超过最大值，则并行度翻倍。

4. 限制并行度

上一部分中描述了数据库引擎如何决定并行度；本部分描述那些由数据库引擎确定的并行度可能被减少的情况。具体来说，并行度减少会在以下情况下发生：

- 启用自适应并行度时
- 启用磁盘I/O上限时
- 数据库资源管理器（Database Resource Manager）限制了并行度时

15

❑ 用户配置文件限制了一个具体的用户可以拥有的并行会话数时

注意，也可以同时启用这些特性中的多个。

警告　只有在满足两个条件的时候，查询优化器才会知晓本部分描述的技术所施加的限制：限制是通过数据库资源管理器施加的，并且使用自动并行度。在其他所有情况下，查询优化器并不知晓有一个限制被施加到并行度上的事实。注意知晓这个信息非常关键，因为由查询优化器估算的成本确实依赖并行度。因此，当不知道限制的存在时，查询优化器可能选择一个不良的执行计划。

● **自适应并行度**

自适应并行度是由parallel_adaptive_multi_user初始化参数控制的。它的用途是影响分配给一个服务进程的从属进程数量。它接受以下两个值。

❑ FALSE：如果进程池没有被耗尽，则从属进程会被按请求的数量分配给服务进程。

❑ TRUE：随着已经分配的从属进程数量的增加，请求的并行度会被自动减小，即使池中可能仍有足够的从属进程满足请求的并行度。此为默认值。

注意　parallel_adaptive_multi_user初始化参数只在两种情况下起作用：第一，当使用手动并行度的时候；第二，将parallel_degree_policy初始化参数设置为limited的情况下使用自动并行度的时候。

为了演示parallel_adaptive_multi_user初始化参数的影响，我们来看一下当以很短的间隔执行不断增加的并发并行操作时分配的从属进程数量。为了这个目的，使用了下面的shell脚本。它的用途是以5秒为间隔启动20个并发的并行度为16（这是在表级别的默认值）的并行查询（每个查询运行持续十几分钟）：

```
sql="select * from t;"
for i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
do
sqlplus -s $user/$password <<<$sql &
sleep 5
done
```

图15-9总结了在11.2版本中测量的结果。通过将parallel_adaptive_multi_user初始化参数设置为FALSE，从属进程数量在达到由parallel_max_servers初始化参数的默认值施加的限制（在本例中是160）之前都随着执行的并行操作数量成比例分配（换句话说，每一个操作都运行在相同的并行度下）。通过将parallel_adaptive_multi_user初始化参数设置为TRUE，从9个并发的并行操作开始，并行度下降了，因此，分配的从属进程数量比请求的要少。

● **磁盘I/O上限**

磁盘I/O上限是从11.1版本开始可用的一项特性。它的用途是根据磁盘I/O子系统能够支撑的最大吞吐率来限制默认的并行度。我强调一下，它只限制默认并行度。因此，如果与默认并行度无关（例

如，当通过指定一个特殊的值使用手动并行度时），磁盘I/O上限根本不会产生影响。

磁盘I/O上限对那些因为不均衡配置导致的I/O受限的系统尤为有用。在并行处理的情况下，一个不均衡的配置经常意味着CPU内核的数量相对于磁盘I/O子系统能够支撑的吞吐率来说太高了。

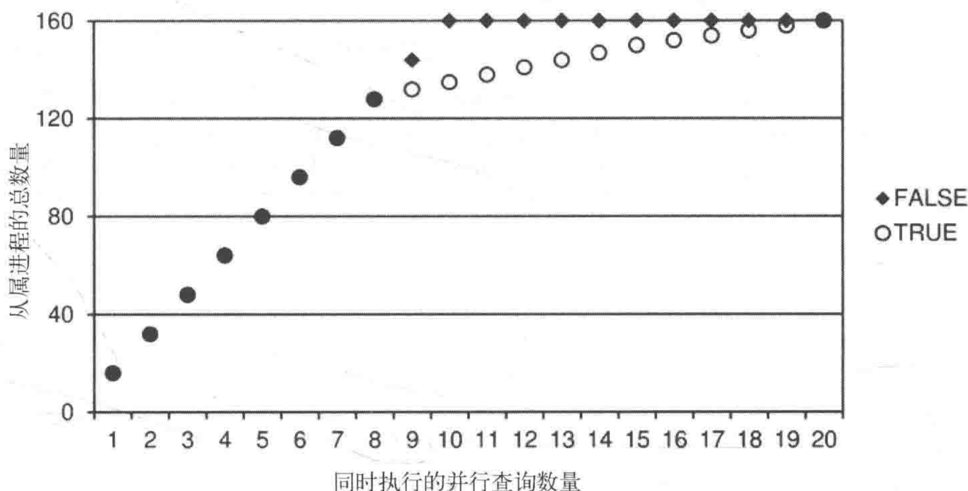


图15-9 *parallel_adaptive_multi_user*初始化参数的影响

提示 对于打算支持大量并行SQL语句的数据库服务器（例如，一个典型的用于数据仓库的数据库服务器），合理的配置是磁盘I/O子系统能支撑的吞吐率应该等于CPU内核数量乘以200 MB/s。例如，如果一个数据库服务器有16个CPU内核，它的磁盘I/O子系统应该支撑的吞吐率为3200 MB/s。

要使用磁盘I/O上限，必须满足两个条件。第一，该特性必须通过一个初始化参数启用。具体是哪一个取决于你使用的版本。

- 在11.1版本中，必须将parallel_io_cap_enabled初始化参数设置为TRUE（默认值是FALSE）。
- 从11.2版本开始，必须将parallel_degree_limit初始化参数设置为IO（默认值是CPU）。注意，从11.2版本开始，应该避免使用parallel_io_cap_enabled初始化参数，因为它是不赞成使用的。

第二，通过I/O口径收集的统计信息必须能够被访问。需要这些统计信息是因为他们向数据库引擎提供关于磁盘I/O子系统能够支撑的最大吞吐率的信息。要收集它们，必须执行dbms_resource_manager包中的calibrate_io过程。下面的PL/SQL代码块，是一段来自px_calibrate_io.sql脚本的摘录，展示了如何执行该过程。注意必须将num_physical_disks参数设置为数据库存储所在的物理磁盘数量（在我的测试系统上，我拥有通过ASM分配的十块磁盘）。

```
DECLARE
  l_max_iops PLS_INTEGER;
  l_max_mbps PLS_INTEGER;
  l_actual_latency PLS_INTEGER;
BEGIN
```

```
  dbms_resource_manager.calibrate_io(
    num_physical_disks => 10,
    max_iops           => l_max_iops,
    max_mbps           => l_max_mbps,
    actual_latency     => l_actual_latency
  );
END;
```

统计信息的收集会持续几分钟。一旦此项工作结束，统计信息的结果就可以通过查询dba_rsrc_io_calibrate视图呈现出来。有两个值与磁盘I/O上限有关：磁盘I/O子系统能够支撑的最大吞吐率（max_mbps）和一个单独的服务进程能够支撑的最大吞吐率（max_pmbps）。在我的测试系统上，它们的值如下：

```
SQL> SELECT max_mbps, max_pmbps
  2  FROM dba_rsrc_io_calibrate;

MAX_MBPS MAX_PMBPS
-------- ---------
     664       297
```

根据上面的查询返回的两个值，数据库引擎计算最大的并行度，如公式15-3所示。如果结果值低于默认的并行度，它就成为新的默认值。如果结果值高于默认的并行度，就会忽略它。

公式15-3　默认并行度受到磁盘I/O子系统能够支撑的最大吞吐率与一个单独的服务进程能够支撑的最大吞吐率之间的比率的限制

$$max_default_dop = \frac{max_mbps}{max_pmbps}$$

- **数据库资源管理器**

资源管理器（Resource Manager）对分配给服务进程的数据库资源提供控制。除了其他功能，可以使用它将并行度限制为一个具体的值。因为描述资源管理器的细节超出本章的范围（参考*Oracle Database Administrator's Guide*手册获取更多信息），我这里只通过px_rm_cap_dop.sql脚本提供一个例子。这个例子展示如何配置资源管理器，以便将某个具体的用户所执行的SQL语句并行度限制为8。配置步骤如下所示。

(1) 创建一个名为control_dop的资源计划（resource plan），通过cap_dop消费者组（consumer group），将并行度限制为8：

```
BEGIN
  dbms_resource_manager.create_pending_area();
  dbms_resource_manager.create_plan(
    plan    => 'CONTROL_DOP',
    comment => 'Control the degree of parallelism'
  );
  dbms_resource_manager.create_consumer_group (
    consumer_group => 'CAP_DOP',
    comment        => 'Users with a restricted degree of parallelism'
  );
  dbms_resource_manager.create_plan_directive(
```

```
    plan                      => 'CONTROL_DOP',
    group_or_subplan          => 'CAP_DOP',
    comment                   => 'Cap degree of parallelism',
    parallel_degree_limit_p1 => 8
  );
  dbms_resource_manager.create_plan_directive(
    plan             => 'CONTROL_DOP',
    group_or_subplan => 'OTHER_GROUPS',
    comment          => 'Unrestricted degree of parallelism'
  );
  dbms_resource_manager.validate_pending_area();
  dbms_resource_manager.submit_pending_area();
END;
```

(2) 提供一个具有切换到cap_dop消费者组权限的特定用户：

```
BEGIN
  dbms_resource_manager_privs.grant_switch_consumer_group(
    grantee_name    => 'CHRIS',
    consumer_group  => 'CAP_DOP',
    grant_option    => FALSE
  );

END;
```

(3) 将一个具体用户的会话与cap_dop消费者组映射：

```
BEGIN
  dbms_resource_manager.create_pending_area();
  dbms_resource_manager.set_consumer_group_mapping(
    attribute      => 'ORACLE_USER',
    value          => 'CHRIS',
    consumer_group => 'CAP_DOP'
  );
  dbms_resource_manager.submit_pending_area();
END;
```

(4) 在系统级别启用control_dop资源计划：

```
ALTER SYSTEM SET resource_manager_plan = control_dop
```

● **用户配置文件**

通过用户配置文件（user profile），尤其是sessions_per_user参数，可以对具体的某个用户能够拥有的并发会话数量做出限制。例如，下面的SQL语句创建一个新的用户配置文件（limit_dop），将会话的数量限制为16，将它与一个用户关联，并且通过将resource_limit初始化参数设置为TRUE（默认值是FALSE）来启用它：

```
CREATE PROFILE limit_dop LIMIT sessions_per_user 16

ALTER USER chris PROFILE limit_dop
```

尽管实际上由用户配置文件施加的限制，最初的引入是为了防止最终用户以超过某个指定值的数量并发登录同一个数据库实例，该限制也可以帮助管理并行会话的数量。这些并发的会话是由数据库

引擎在一条SQL语句并行执行的时候自动创建的。该限制也适用于它们。

额外的会话被创建出来的原因是，一条并行执行的SQL语句不仅需要一个查询协调器的会话，而且每个从属进程也需要一个会话。结果，即使一个最终用户只登录了一次，他也可能请求多个会话。因此，取决于sessions_per_user参数是如何设置的，并行度可能会受到限制。

提示　我不建议使用用户配置文件限制资源。要进行限制，应该尽量使用资源管理器。我介绍用户配置文件方法仅仅是因为你需要知道用户配置文件可以限制并行度。

---

5. 降级

当查询协调器请求的从属进程数量高于它实际可以获得的从属进程数量的时候就会发生降级。以下两种情况下会发生降级。

- 当并行度受到上一部分中描述的技术限制的时候。换句话说，当并行度受到自适应并行度、资源管理器（仅对于手动并行度来说）或用户配置文件限制的时候。
- 当查询协调器从池中请求的从属进程数量高于实际可用的从属进程数量的时候。

事实上，在查询协调器请求某一数量的从属进程时，取决于已经在运行中的从属进程有多少，数据库引擎可能无法满足该请求。例如，如果从属进程的最大数量设置为40，对于图15-7中所示的执行计划来说只有5个并发的SQL语句（40/8）能够通过所请求的并行度（请求8个从属进程）执行。当达到上限时，有以下三种可能性。

- 并行度被降级（换句话说，减少并行度）。
- 将一个 ORA-12827: insufficient parallel query slaves available 错误返回给查询协调器。
- 或者该SQL语句的执行计划被置于保持状态，直到所需数量的从属进程可用。

后面的方法仅用于从11.2开始的版本，而且仅在启用语句排队（statement queuing）的情况下（下一部分会介绍此内容）。如果没有启用语句排队，就会使用其他两种方法中的一个。必须通过设置parallel_min_percent初始化参数来配置具体使用哪一种方法。可以将该参数设置为一个从0到100之间的整数值。主要情况有以下三种。

- 0：这个值（也就是默认值）指定可以被静默降级的并行度。换句话说，数据库引擎能够提供尽可能多的从属进程。如果可用的从属进程少于两个，执行就会改为串行。这意味着SQL语句总会被执行，永远不会遇到ORA-12827错误。
- 1~99：从1到99范围内的值以百分比形式为降级指定一个限制。必须至少提供达到指定百分比的从属进程；否则，会引发ORA-12827错误。例如，如果将它设置为25而且有会话请求16个从属进程，那么必须至少提供4个（16 × 25/100）可用的从属进程才可以避免此错误。
- 100：使用这个值，要么提供所有请求的从属进程，要么引发ORA-12827错误。

下面的例子（来自px_min_percent.sql脚本），在没有其他并行执行的语句运行的情况下执行，证实了这一点（注意，40是50的80%）：

```
SQL> ALTER SYSTEM SET parallel_max_servers = 40;

SQL> ALTER TABLE t PARALLEL 50;
```

```
SQL> ALTER SESSION SET parallel_min_percent = 80;

SQL> SELECT count(pad) FROM t;

COUNT(PAD)
----------
    100000

SQL> SELECT * FROM table(dbms_xplan.display_cursor(NULL, NULL, 'basic +parallel'));

------------------------------------------------------------------------
| Id  | Operation                | Name     |    TQ  |IN-OUT| PQ Distrib |
------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |          |        |      |            |
|   1 |  SORT AGGREGATE          |          |        |      |            |
|   2 |   PX COORDINATOR         |          |        |      |            |
|   3 |    PX SEND QC (RANDOM)   | :TQ10000 |  Q1,00 | P->S | QC (RAND)  |
|   4 |     SORT AGGREGATE       |          |  Q1,00 | PCWP |            |
|   5 |      PX BLOCK ITERATOR   |          |  Q1,00 | PCWC |            |
|   6 |       TABLE ACCESS FULL| T          |  Q1,00 | PCWP |            |
------------------------------------------------------------------------

SQL> ALTER SESSION SET parallel_min_percent = 81;

SQL> SELECT count(pad) FROM t;
SELECT count(pad) FROM t
*
ERROR at line 1:
ORA-12827: insufficient parallel query slaves (requested 50, available 40, parallel_min_percent 81)
```

如果想要知道在一个运行中的数据库实例上有多少个操作被降级以及降级了多少，可以执行下面的查询。显然，当你看见太多的降级，尤其是当很多操作被改成串行时，应该会怀疑配置的问题了：

```
SQL> SELECT name, value
  2  FROM v$sysstat
  3  WHERE name like 'Parallel operations%';

NAME                                          VALUE
-------------------------------------------- -----
Parallel operations not downgraded              14
Parallel operations downgraded to serial        10
Parallel operations downgraded 75 to 99 pct     14
Parallel operations downgraded 50 to 75 pct      2
Parallel operations downgraded 25 to 50 pct      0
Parallel operations downgraded 1 to 25 pct       0
```

6. 语句排队

语句排队是一项从11.2版本开始可用的特性，旨在避免降级。要达到这一目标，资源管理器要在没有足够可用的从属进程来执行具体的一条SQL语句时识别到问题。当资源管理器识别到这样的情况，它就会通过使会话在一个等待列表中排队来挂起该执行。然后，一旦请求数量的从属进程可用，它就会使会话出队并恢复执行。默认情况下，该等待列表是通过一个先进先出的队列管理的，在有可

用的从属进程之前会话必须等待（换句话说，这里没有超时的概念）。

可能与你预期的相反，资源管理器不会通过比较当前活跃的从属进程数量和数据库引擎能够处理的最大数量（这个值通过parallel_max_servers初始化参数定义）来检查是否有足够数量的从属进程可用。相反，资源管理器使用另一个通过parallel_servers_target初始化参数配置的阈值。使用这个额外的初始化参数背后的原因非常简单：没有必要为所有并行执行的SQL语句启用语句排队。因此，通过语句排队来管理仅一部分的从属进程池可能是值得的。

尽管parallel_servers_target初始化参数是动态的，它也仅能够在系统级别进行设置。此外，在12.1多租户环境下，无法在PDB级别设置它。换句话说，对于从属进程池来说，该配置仅能在数据库实例级别执行。

语句排队仅在两种情况下启用。第一，对于那些已将parallel_degree_policy初始化参数被置为auto的会话执行的SQL语句（如果它们不包含一个禁用语句排队的hint）。第二，对于那些包含statement_queuing hint的SQL语句。第二种可能性作用于使用手动并行度的应用程序。

对于那些已将parallel_degree_policy初始化参数被置为auto的会话来说，语句排队可以在SQL语句级别通过添加no_statement_queuing hint显式禁用。

当一个会话在队列中等待时，它的等待事件是resmgr:pq queued。例如，下面的查询展示哪些会话因为语句排队而被挂起了：

```
SQL> SELECT sid, seconds_in_wait
  2  FROM v$session
  3  WHERE event = 'resmgr:pq queued';

SID SECONDS_IN_WAIT
--- ---------------
113             121
 37              60
143              34
```

挂起的会话也可以通过v$rsrc_session_info动态性能视图来监控。使用它，不仅可以看见它们在队列中等待了多长时间（current_pq_queued_time列，以毫秒为单位），还可以知道下一个出队的会话是哪一个（pq_status列被设置为Queue head）以及每个挂起的会话请求的从属进程数量（pq_servers列）是多少。注意，后面两个列仅从12.1版本开始可用。下面的例子为上面查询中已经列举的三个会话证实了这一点：

```
SQL> SELECT sid, pq_status, current_pq_queued_time, pq_servers
  2  FROM v$rsrc_session_info
  3  WHERE state = 'PQ QUEUED';

SID PQ_STATUS  CURRENT_PQ_QUEUED_TIME PQ_SERVERS
--- ---------- ---------------------- ----------
113 Queue head                 121068         16
 37 Queued                      60686         16
143 Queued                      34843          4
```

除了刚刚描述的默认行为以外，资源管理器还提供几个指令，使用这些指令可以设置利用以下特性的资源计划：

- ❑ 在等待列表中管理会话的出队顺序

- 在消费组级别限制从属进程的使用
- 指定超时时间
- 定义关键SQL语句以绕过语句排队（仅12.1版本）
- 在多租户环境下管理语句排队（仅12.1版本）

关于这些特性的详细信息在*Oracle Database VLDB and Partitioning Guide*手册中提供。

7. 并行查询

以下这些操作，在查询和子查询中，都可以使用并行方式执行：

- 全表扫描、全分区扫描以及索引快速全扫描
- 索引全扫描和范围扫描，但是前提是索引是分区的（在一个给定时间，一个分区只能同时被一个从属进程访问，其副作用是，并行度会受到访问的分区数量的限制）
- 联接（第14章也提供了一些例子）
- 集合操作符
- 排序
- 聚合

注意 全表扫描、全分区扫描以及索引快速全扫描在并行执行时通常会使用直接路径读，因此，会绕过缓冲区缓存。一个例外是从11.2版本开始，当激活内存中的并行执行的时候。事实上，内存中执行的目标恰好是避免直接路径读并缓存尽可能多的数据。注意对于索引全扫描和范围扫描，数据库引擎总是执行正常的物理读。

当查询引用了不支持并行处理的用户自定义函数时，它们无法通过并行方式执行。基本上，要支持并行处理，用户自定义函数必须既不能写入数据库也不能读取或修改包变量。当编写PL/SQL代码时，应该使用PARALLEL_ENABLE子句标记支持并行处理的用户自定义函数。注意在某些情况下用户定义函数并行执行的能力不仅取决于用户自定义函数本身，而且还取决于调用的查询（而且，最终取决于执行计划本身）。px_query_udf.sql脚本提供了一个例子，展示一个函数阻止一个查询以并行方式运行，而在另一个查询上却不会施加这样的限制。

并行查询默认是启用的。在会话级别，可以使用下面的SQL语句启用或禁用它们：

```
ALTER SESSION ENABLE PARALLEL QUERY

ALTER SESSION DISABLE PARALLEL QUERY
```

此外，也可以在启用并行查询的同时，覆盖由段级别的手动并行度或使用以下SQL语句的自动并行度所定义的并行度：

```
ALTER SESSION FORCE PARALLEL QUERY PARALLEL 4
```

然而，要记住hint要优先于会话级别的设置。一方面，即使在会话级别禁用了并行查询，hint也可以启用一个并行执行。真正关闭并行查询的方法只有两个，将parallel_max_servers初始化参数设置为0，或者通过配置资源管理器来关闭。另一方面，即使在会话级别强制指定一个并行度，hint也能引导为另一个并行度。要检查在会话级别是否启用了并行查询，可以执行一条类似下面这样的查询

（pq_status列被设置为ENABLED、DISABLEOD或FORCED）：

```
SELECT pq_status
FROM v$session
WHERE sid = sys_context('userenv','sid')
```

下面的执行计划展示一个使用并行索引范围扫描、并行全表扫描以及并行散列联接的例子。它来自px_query.sql脚本。注意这些hint：parallel_index hint用于索引访问，而parallel hint用于表扫描。两个hint都是使用对象级别语法将并行度指定为2。此外，pq_distribute hint用于指定分配方法。列TQ包含三个值，也就意味着有三组从属进程用于实施这个执行计划。操作8以并行方式扫描索引i1（这是可行的，因为索引是分区的）。然后，操作7，使用从索引i1提取的rowid，访问表t1。如在操作6中所示，分区粒度被用于这两个操作。接下来，数据被使用散列分布发送给消费者（组Q1,02的从属进程）。当消费者接收数据（操作4）后，他们将数据传递给操作3以在内存中为散列联接构建散列表。一旦表t1的数据全部被处理完毕，表t2的并行全扫描就可以开始了。这是在操作12中执行的。如在操作11中所示，块范围粒度被用于此操作。然后数据被通过散列分布发送给消费者（组Q1,02的从属进程）。当消费者接收到数据（操作9）后，他们将数据传递给操作3以探测该散列表。最后，操作2将满足联接条件的数据发送给查询协调器（图15-10演示了这个执行计划）：

```
SELECT /*+ leading(t1) use_hash(t2)
           index(t1) parallel_index(t1 2)
           full(t2) parallel(t2 2)
           pq_distribute(t2 hash,hash) */ *
FROM t1, t2
WHERE t1.id > 9000
AND t1.id = t2.id+1

-----------------------------------------------------------------------------------------------
| Id | Operation                    | Name     | Pstart| Pstop |   TQ  |IN-OUT| PQ Distri |
-----------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT             |          |       |       |       |      |           |
|  1 |  PX COORDINATOR              |          |       |       |       |      |           |
|  2 |   PX SEND QC (RANDOM)        | :TQ10002 |       |       | Q1,02 | P->S | QC (RAND) |
|* 3 |    HASH JOIN BUFFERED        |          |       |       | Q1,02 | PCWP |           |
|  4 |     PX RECEIVE               |          |       |       | Q1,02 | PCWP |           |
|  5 |      PX SEND HASH            | :TQ10000 |       |       | Q1,00 | P->P | HASH      |
|  6 |       PX PARTITION HASH ALL  |          |     1 |     4 | Q1,00 | PCWC |           |
|  7 |        TABLE ACCESS BY INDEX ROW| T1    |       |       | Q1,00 | PCWP |           |
|* 8 |         INDEX RANGE SCAN     | I1       |     1 |     4 | Q1,00 | PCWP |           |
|  9 |     PX RECEIVE               |          |       |       | Q1,02 | PCWP |           |
| 10 |      PX SEND HASH            | :TQ10001 |       |       | Q1,01 | P->P | HASH      |
| 11 |       PX BLOCK ITERATOR      |          |       |       | Q1,01 | PCWC |           |
|*12 |        TABLE ACCESS FULL     | T2       |       |       | Q1,01 | PCWP |           |
-----------------------------------------------------------------------------------------------

  3 - access("T1"."ID"="T2"."ID"+1)
  8 - access("T1"."ID">9000)
 12 - filter("T2"."ID"+1>9000)
```

根据执行计划和图15-10，使用了一个数据流操作和三个表队列。注意，即使图15-10显示了三组从属进程（因为请求的并行度是2，所以一共有6个从属进程），在执行期间，只有两组是从池中分配

的（换句话说，四个从属进程）。这是因为一个单独的数据流操作无法使用超过两组的从属进程。这个特殊的例子中会出现的情况是用于扫描表t1（Q1,00）的那一组永远不会与扫描表t2（Q1,01）的那一组同时并发运行。因此，查询协调器简单地为这两组（重复）使用相同的从属进程。

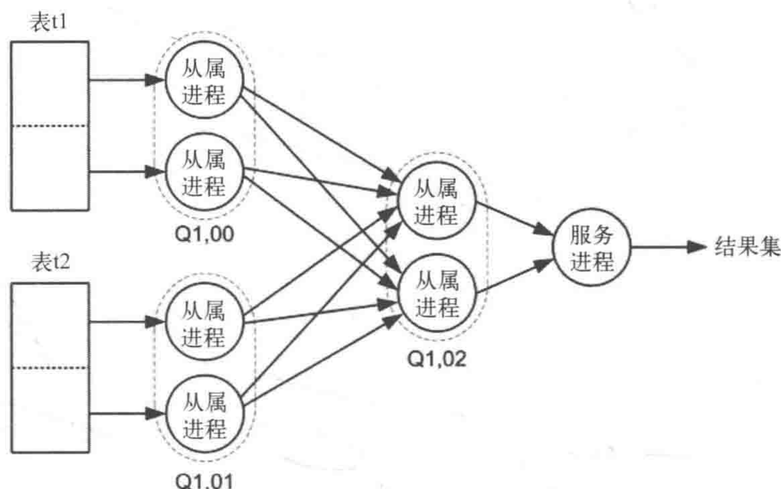


图15-10　尽管出现了三组从属进程，一个单独的数据流操作也无法使用超过两组的从属进程执行

警告　HASH JOIN BUFFERED操作（在上面的执行计划中，操作3）不仅创建一张包含构建的输入（操作5到8）返回的数据的散列表，而且还会缓存由满足联接条件的探测输入（操作10到操作12）返回的数据。因此，就有了后缀BUFFERED。这是数据库引擎因为内部限制（两种分布操作无法在同一时间被激活）不得不实现的一种特殊行为。从性能的角度来看，缓冲可能会是一个重大问题。

---

8. 并行DML语句

以下DML语句可以使用并行方式执行：

- ❑ DELETE
- ❑ 带有子查询的INSERT（带有VALUES子句的INSERT语句无法被并行化）
- ❑ MERGE
- ❑ UPDATE

注意　INSERT语句和MERGE语句（对于插入数据的部分）并行执行的时候使用直接路径插入。因此，它们不仅受到直接路径插入的优势和劣势的影响，而且也受到直接路径插入的限制条件的制约。我会在15.4节介绍它们。

在以下情况下DML语句无法通过并行方式执行：

- ❑ 某张表上有触发器；
- ❑ 某张表有一个引用自身的外键约束，或者有一个带有级联删除的外键约束，或者有一个延迟约束；
- ❑ 它们是在一个分布式事务中执行的；
- ❑ 它们引用了一个远程对象；
- ❑ 它们引用了一个无法并行执行的用户自定义函数（在PL/SQL中，使用PARALLEL_ENABLE子句来标记支持并行处理的函数）；
- ❑ 修改了某个对象列；
- ❑ 或修改了某个群集或临时表。

并行DML语句默认是禁用的（小心，此处与并行查询正好相反）。在会话级别，可以通过以下的SQL语句启用和禁用它们：

```
ALTER SESSION ENABLE PARALLEL DML

ALTER SESSION DISABLE PARALLEL DML
```

此外，也可以通过以下SQL语句强制并行执行来使用某个具体的并行度：

```
ALTER SESSION FORCE PARALLEL DML PARALLEL 4
```

自12.1版本起，还可能用enable_parallel_dml和disable_parallel_dml hint来启用和禁用并行DML语句。与并行查询会出现的情况形成对照的是，只使用parallel和parallel_index hint无法启用并行DML语句。换句话说，DML语句如果要利用并行处理只能在会话级别或SQL语句级别启用。要检查并行DML语句在会话级别是处于启用还是禁用状态，可以执行类似以下的查询（pdml_status列被设置为ENABLED、DISABLED或FORCED）：

```
SELECT pdml_status
FROM v$session
WHERE sid = sys_context('userenv','sid')
```

除了INSERT语句，必须同时启用并行查询以支持使用并行方式执行DML语句。事实上，DML语句基本是由两个操作组成：首先找到要修改的数据，然后第二步才是修改它们。问题是如果查找数据的部分不是以并行方式执行的，则无法并行化修改数据的部分。为了验证这种行为，我们来看几个来自px_dml.sql脚本的例子。

- ❑ 仅启用并行DML语句时，没有一个操作是并行化的：

```
SQL> ALTER SESSION DISABLE PARALLEL QUERY;

SQL> ALTER SESSION ENABLE PARALLEL DML;

SQL> ALTER TABLE t PARALLEL 2;

SQL> UPDATE t SET id = id + 1;

-----------------------------------
| Id  | Operation          | Name |
-----------------------------------
|   0 | UPDATE STATEMENT   |      |
|   1 |  UPDATE            | T    |
|   2 |   TABLE ACCESS FULL| T    |
-----------------------------------
```

- 仅启用并行查询时，DML语句的更新部分没有以并行方式执行。事实上，只有操作3到操作5是由从属进程执行的。因此，更新部分（操作1）是由查询协调器串行执行的：

```
SQL> ALTER SESSION ENABLE PARALLEL QUERY;

SQL> ALTER SESSION DISABLE PARALLEL DML;

SQL> ALTER TABLE t PARALLEL 2;

SQL> UPDATE t SET id = id + 1;

-----------------------------------------------------------------------
| Id  | Operation             | Name     |    TQ  |IN-OUT| PQ Distrib |
-----------------------------------------------------------------------
|   0 | UPDATE STATEMENT      |          |        |      |            |
|   1 |  UPDATE               | T        |        |      |            |
|   2 |   PX COORDINATOR      |          |        |      |            |
|   3 |    PX SEND QC (RANDOM)| :TQ10000 |  Q1,00 | P->S | QC (RAND)  |
|   4 |     PX BLOCK ITERATOR |          |  Q1,00 | PCWC |            |
|   5 |      TABLE ACCESS FULL| T        |  Q1,00 | PCWP |            |
-----------------------------------------------------------------------
```

- 同时启用并行查询和并行DML语句时，更新部分（操作3）以并行方式执行。在此情况下，只使用了一组从属进程（操作2到操作5在TQ列上拥有相同的值）。这暗示每个从属进程扫描各自的表分区粒度并修改它所找到的记录：

```
SQL> ALTER SESSION ENABLE PARALLEL QUERY;

SQL> ALTER SESSION ENABLE PARALLEL DML;

SQL> ALTER TABLE t PARALLEL 2;

SQL> UPDATE t SET id = id + 1;

-----------------------------------------------------------------------
| Id  | Operation             | Name     |    TQ  |IN-OUT| PQ Distrib |
-----------------------------------------------------------------------
|   0 | UPDATE STATEMENT      |          |        |      |            |
|   1 |  PX COORDINATOR       |          |        |      |            |
|   2 |   PX SEND QC (RANDOM) | :TQ10000 |  Q1,00 | P->S | QC (RAND)  |
|   3 |    UPDATE             | T        |  Q1,00 | PCWP |            |
|   4 |     PX BLOCK ITERATOR |          |  Q1,00 | PCWC |            |
|   5 |      TABLE ACCESS FULL| T        |  Q1,00 | PCWP |            |
-----------------------------------------------------------------------
```

9. 并行DDL语句

表和索引支持并行DDL语句。以下是三个有代表性的并行化操作：

- CREATE TABLE ... AS SELECT ...（CTAS）语句
- 索引的创建和重建
- 约束的创建和验证

此外，对于已分区表和索引，还可以并行化类似COALESCE、MOVE和SPLIT这样的分区管理操作。通

常，可利用并行处理的DDL语句会提供PARALLEL子句（很快你就会看到，约束是一个例外）来指定是否应该使用并行处理，以及如果使用了并行处理，其并行度是多少。

默认会启用并行DDL语句。在会话级别，可以使用下面的SQL语句启用或禁用它们：

```
ALTER SESSION ENABLE PARALLEL DDL

ALTER SESSION DISABLE PARALLEL DDL
```

也可以通过使用下面的SQL语句以某个特定的并行度强制并行执行（对于支持它的DDL语句来说）：

```
ALTER SESSION FORCE PARALLEL DDL PARALLEL 4
```

要检查是否在会话级别启用或禁用了并行DDL语句，可以执行类似下面的查询（pddl_status列被设置为ENABLED、DISABLED或FORCED）：

```
SELECT pddl_status
FROM v$session
WHERE sid = sys_context('userenv','sid')
```

对于可以并行执行的这三种主要DDL语句类型，接下来的几个部分将会展示几个基于px_ddl.sql脚本的例子。

- **CTAS语句**

一条CTAS语句是由两个处理数据的操作组成：用于从来源表中检索数据的查询操作和向目标表插入数据的插入操作。每个部分都可以独立于彼此使用串行或并行方式执行。但是，如果使用了并行处理，一般会将两个操作都并行化。下面的执行计划演示了这一点。

- 插入并行化：只有操作2至操作4是以并行方式执行的。查询协调器扫描表t1并使用循环方法（round-robin）将它的内容分发给从属进程。既然查询协调器和多个从属进程通信，这些操作之间的关系就是串行到并行（S->P）。一组从属进程接收这些数据并以并行方式执行插入（操作LOAD AS SELECT）：

```
CREATE TABLE t2 PARALLEL 2 AS SELECT /*+ no_parallel(t1) */ * FROM t1

-----------------------------------------------------------------------
| Id  | Operation              | Name     |    TQ  |IN-OUT| PQ Distrib |
-----------------------------------------------------------------------
|   0 | CREATE TABLE STATEMENT |          |        |      |            |
|   1 |  PX COORDINATOR        |          |        |      |            |
|   2 |   PX SEND QC (RANDOM)  | :TQ10001 |  Q1,01 | P->S | QC (RAND)  |
|   3 |    LOAD AS SELECT      | T2       |  Q1,01 | PCWP |            |
|   4 |     PX RECEIVE         |          |  Q1,01 | PCWP |            |
|   5 |      PX SEND ROUND-ROBIN| :TQ10000 |       | S->P | RND-ROBIN  |
|   6 |       TABLE ACCESS FULL | T1       |       |      |            |
-----------------------------------------------------------------------
```

- 查询并行化：只有操作3至操作5是以并行方式执行的。从属进程基于块范围粒度以并行方式扫描表t1并将其内容发送给查询协调器，这就是并行到串行（P->S）关系的原因。查询协调器执行插入（操作LOAD AS SELECT）：

```
CREATE TABLE t2 NOPARALLEL AS SELECT /*+ parallel(t1 2) */ * FROM t1
```

```
-----------------------------------------------------------------------------
| Id  | Operation              | Name     |    TQ  |IN-OUT| PQ Distrib |
-----------------------------------------------------------------------------
|   0 | CREATE TABLE STATEMENT |          |        |      |            |
|   1 |  LOAD AS SELECT        | T2       |        |      |            |
|   2 |   PX COORDINATOR       |          |        |      |            |
|   3 |    PX SEND QC (RANDOM) | :TQ10000 |  Q1,00 | P->S | QC (RAND)  |
|   4 |     PX BLOCK ITERATOR  |          |  Q1,00 | PCWC |            |
|   5 |      TABLE ACCESS FULL | T1       |  Q1,00 | PCWP |            |
-----------------------------------------------------------------------------
```

- 两个操作全部并行化：从属进程基于块范围粒度以并行方式扫描表t1并直接将它们获得的数据插入到目标表中，不需要将数据发送给另一个并行从属组。应该注意两件重要的事情，第一，查询协调器并没有直接参与数据的处理。第二，数据并没有通过表队列发送（除了由操作2发送给查询协调器的少量信息，没有发生任何通信）：

```
CREATE TABLE t2 PARALLEL 2 AS SELECT /*+ parallel(t1 2) */ * FROM t1

-----------------------------------------------------------------------------
| Id  | Operation              | Name     |    TQ  |IN-OUT| PQ Distrib |
-----------------------------------------------------------------------------
|   0 | CREATE TABLE STATEMENT |          |        |      |            |
|   1 |  PX COORDINATOR        |          |        |      |            |
|   2 |   PX SEND QC (RANDOM)  | :TQ10000 |  Q1,00 | P->S | QC (RAND)  |
|   3 |    LOAD AS SELECT      | T2       |  Q1,00 | PCWP |            |
|   4 |     PX BLOCK ITERATOR  |          |  Q1,00 | PCWC |            |
|   5 |      TABLE ACCESS FULL | T1       |  Q1,00 | PCWP |            |
-----------------------------------------------------------------------------
```

上面的例子展示了实用对象级别语法的hint。要为两个操作都启用并行化（最后一个例子），从11.2版本开始，对于CTAS语句可以使用语句级别的语法，如下例所示：

```
CREATE /*+ parallel(2) */ TABLE t2 AS SELECT * FROM t1
```

与之前的那一种相比，使用这种语法的一个重要的区别就是，因为没有指定PARALLEL子句，并行度只用于表的创建阶段。换句话说，在数据字典中，与这张表关联的并行度是1。

- **索引的创建和重建**

可以通过并行方式创建和重建索引。要完成创建过程，需要两组从属进程合作。第一组读取被索引的数据。第二组对其从第一组接收的数据进行排序并建立索引。下面的SQL语句是一个例子。注意第一组如何执行操作6到操作8（Q1,00）以及第二组如何执行操作2到操作5（Q1,01）。数据是以范围方法在这两组进程之间进行分布的（所以第二组的每个并行子进程处理它分配到的索引的一小部分），而且拥有一个并行到并行（P->P）的关系：

```
CREATE INDEX i1 ON t1 (id) PARALLEL 4

-----------------------------------------------------------------------------
| Id  | Operation              | Name     |    TQ  |IN-OUT| PQ Distrib |
-----------------------------------------------------------------------------
|   0 | CREATE INDEX STATEMENT |          |        |      |            |
```

15

```
|   1 |  PX COORDINATOR          |          |        |      |            |
|   2 |   PX SEND QC (ORDER)     | :TQ10001 |  Q1,01 | P->S | QC (ORDER) |
|   3 |    INDEX BUILD NON UNIQUE| I1       |  Q1,01 | PCWP |            |
|   4 |     SORT CREATE INDEX    |          |  Q1,01 | PCWP |            |
|   5 |      PX RECEIVE          |          |  Q1,01 | PCWP |            |
|   6 |       PX SEND RANGE      | :TQ10000 |  Q1,00 | P->P | RANGE      |
|   7 |        PX BLOCK ITERATOR |          |  Q1,00 | PCWC |            |
|   8 |         TABLE ACCESS FULL| T1       |  Q1,00 | PCWP |            |
---------------------------------------------------------------------------
```

索引的重建也会引导出十分类似的执行计划（注意根据操作8，数据是从索引中提取的，而不是从表中）：

```
ALTER INDEX i1 REBUILD PARALLEL 4

------------------------------------------------------------------------------
| Id  | Operation                   | Name      |     TQ |IN-OUT| PQ Distrib |
------------------------------------------------------------------------------
|   0 | ALTER INDEX STATEMENT       |           |        |      |            |
|   1 |  PX COORDINATOR             |           |        |      |            |
|   2 |   PX SEND QC (ORDER)        | :TQ10001  |  Q1,01 | P->S | QC (ORDER) |
|   3 |    INDEX BUILD NON UNIQUE   | I1        |  Q1,01 | PCWP |            |
|   4 |     SORT CREATE INDEX       |           |  Q1,01 | PCWP |            |
|   5 |      PX RECEIVE             |           |  Q1,01 | PCWP |            |
|   6 |       PX SEND RANGE         | :TQ10000  |  Q1,00 | P->P | RANGE      |
|   7 |        PX BLOCK ITERATOR    |           |  Q1,00 | PCWC |            |
|   8 |         INDEX FAST FULL SCAN| I1        |  Q1,00 | PCWP |            |
------------------------------------------------------------------------------
```

● **约束的创建和验证**

创建或验证约束（例如外键和检查约束）时，必须验证已经存储在表中的数据。出于这个目的，数据库引擎执行一个递归查询。例如，假设执行下面的SQL语句：

```
ALTER TABLE t ADD CONSTRAINT t_id_nn CHECK (id IS NOT NULL)
```

数据库引擎递归地执行类似下面这样的查询来验证存储在表中的数据（注意，如果查询返回结果为空，数据就是有效的）：

```
SELECT rowid
FROM t
WHERE NOT (id IS NOT NULL)
```

因此，如果约束创建所属的表拥有值为2或更高的并行度，数据库引擎会以并行方式执行该查询。

注意 在表级别定义的并行度用于递归查询，与在会话级别是否启用、禁用或强制并行查询和并行DDL语句没有关系。换句话说，ALTER SESSION ... PARALLEL语句对递归查询没有影响。

定义主键约束时，数据库引擎无法以并行方式创建索引。为避免这个限制，必须在定义约束之前创建（唯一）索引。下面的SQL语句进行了演示：

```
CREATE UNIQUE index t_pk ON t (id) PARALLEL 2

ALTER TABLE t ADD CONSTRAINT t_pk PRIMARY KEY (id)
```

15.3.2　何时使用

并行处理只有在满足两个条件时才应该被使用。第一，在有大量可用的空闲资源（CPU、内存，及磁盘I/O带宽）的情况下可以使用该技术。记住，并行处理的目标是通过将一个通常由单独的进程（此时也就会使用一个单独CPU内核）所做的工作分散给多个进程（此时就会使用多个CPU内核）以减少响应时间。第二，可以为那些使用串行方式执行超过十几秒钟的SQL语句使用该技术；否则，并行环境（主要是从属进程和表队列）初始化、协调和终止所需的时间及资源，可能会比从并行化本身获益的还要高一些。实际使用时的限制取决于可用的资源总量。因此，在某些情况下，只有那些花费超过几分钟的SQL语句，或甚至更长时间的语句，才适合使用并行执行。值得注意的是，如果这两个条件不满足，性能恐怕会不升反降。

如果经常对很多的SQL语句使用并行处理，应该在系统级别启用自动并行度，或者在段级别启用手动并行度。否则，如果它仅用于特定的批处理或报表，一般来说最好是在会话级别通过hint启用它比较好。

15.3.3　陷阱和谬误

有一点你需要明白，在对象级别语法中使用parallel和parallel_index这两个hint，并不会强制查询优化器使用并行处理。反而，它们会覆盖在表或索引级别上定义的并行度。因此，添加这两个hint会允许查询优化器使用指定的并行度来考虑并行处理。这意味着查询优化器会分别在使用和不使用并行处理的情况下考虑执行计划，并且按照惯例，从中选出具有较低成本的那一个。接下来我会通过展示一个来自px_dop_manual.sql脚本的例子来加深你的印象。如下面的SQL语句所示，与全表扫描关联的成本随着并行度成比例下降（参考第7章，获取更多关于并行操作成本的信息）：

```
SQL> EXPLAIN PLAN SET STATEMENT_ID 'dop1' FOR
  2  SELECT /*+ full(t) parallel(t 1) */ * FROM t WHERE id > 93000;

SQL> EXPLAIN PLAN SET STATEMENT_ID 'dop2' FOR
  2  SELECT /*+ full(t) parallel(t 2) */ * FROM t WHERE id > 93000;

SQL> EXPLAIN PLAN SET STATEMENT_ID 'dop3' FOR
  2  SELECT /*+ full(t) parallel(t 3) */ * FROM t WHERE id > 93000;

SQL> EXPLAIN PLAN SET STATEMENT_ID 'dop4' FOR
  2  SELECT /*+ full(t) parallel(t 4) */ * FROM t WHERE id > 93000;

SQL> SELECT statement_id, cost
  2  FROM plan_table
  3  WHERE id = 0;

STATEMENT_ID COST
------------ ----
dop1          296
```

```
dop2          164
dop3          110
dop4           82
```

如果该SQL语句在没有hint并且并行度设置为1的情况下执行，查询优化器会选择索引范围扫描：

```
SQL> SELECT * FROM t WHERE id > 93000;

-------------------------------------------------------
| Id  | Operation                   | Name | Cost (%CPU)|
-------------------------------------------------------
|   0 | SELECT STATEMENT            |      |   125   (0)|
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |   125   (0)|
|*  2 |   INDEX RANGE SCAN          | I    |    17   (0)|
-------------------------------------------------------

   2 - access("ID">93000)
```

注意，与上面执行计划关联的成本（125）要低于使用并行度为2时的全表扫描成本。相较而言，使用大于或等于3的并行度时，全表扫描的成本更低一些。

现在，我们来看一下仅将`parallel` hint添加到SQL语句时会发生什么，换句话说，就是不使用访问路径hint时。结果是当并行度设置为2的时候，查询优化器选择了串行的索引范围扫描，而当并行度设置为3的时候选择了并行全表扫描：

```
SQL> SELECT /*+ parallel(t 2) */ * FROM t WHERE id > 93000;

-------------------------------------------------------
| Id  | Operation                   | Name | Cost (%CPU)|
-------------------------------------------------------
|   0 | SELECT STATEMENT            |      |   125   (0)|
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |   125   (0)|
|*  2 |   INDEX RANGE SCAN          | I    |    17   (0)|
-------------------------------------------------------

   2 - filter("ID">93000)

SQL> SELECT /*+ parallel(t 3) */ * FROM t WHERE id > 93000;

----------------------------------------------------
| Id  | Operation              | Name     | Cost (%CPU)|
----------------------------------------------------
|   0 | SELECT STATEMENT       |          |   110   (1)|
|   1 |  PX COORDINATOR        |          |            |
|   2 |   PX SEND QC (RANDOM)  | :TQ10000 |   110   (1)|
|   3 |    PX BLOCK ITERATOR   |          |   110   (1)|
|*  4 |     TABLE ACCESS FULL  | T        |   110   (1)|
----------------------------------------------------

   4 - filter("ID">93000)
```

总之，`parallel`和`parallel_index`这两个hint只是简单地允许查询优化器考虑并行处理；它们不会强制查询优化器做什么。

为了让并行化的执行更高效，将工作总量在所有的从属进程中间进行平均分配十分关键。事实上，

所有属于一个组的从属进程，在同组内的所有从属进程都执行完毕之前必须处于等待状态。简而言之，并行操作的速度取决于最慢的那个从属进程。如果想要检查一条SQL语句工作量的实际分布情况，你既可以使用实时监控（参见第4章），也可以使用v$pq_tqstat动态性能视图。基本上，该视图对于每一个从属进程以及执行计划中的每一个PX SEND和PX RECEIVE操作都会显示一行记录。需要注意这个信息只会提供当前会话以及最近以并行方式成功执行的SQL语句。我们来看一个例子，这个例子来自px_tqstat.sql脚本生成的输出。两份输出之间的映射是通过执行计划的TQ列以及v$pq_tqstat视图的dfo_number列和tq_id列来完成的。那么要记住，与之前解释的一样，执行计划显示关于生产者的信息。例如，Q1,00对应dfo_number等于1并且tq_id等于0的记录。此外，PX SEND操作对应生产者，PX RECEIVE操作对应消费者：

```
SQL> SELECT * FROM t t1, t t2 WHERE t1.id = t2.id;

-----------------------------------------------------------------------------------------
| Id  | Operation                  | Name     | Pstart| Pstop |    TQ  |IN-OUT| PQ Distrib |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |          |       |       |        |      |            |
|   1 |  PX COORDINATOR            |          |       |       |        |      |            |
|   2 |   PX SEND QC (RANDOM)      | :TQ10001 |       |       |  Q1,01 | P->S | QC (RAND)  |
|*  3 |    HASH JOIN               |          |       |       |  Q1,01 | PCWP |            |
|   4 |     PX RECEIVE             |          |       |       |  Q1,01 | PCWP |            |
|   5 |      PX SEND PARTITION (KEY)| :TQ10000 |       |       |  Q1,00 | P->P | PART (KEY) |
|   6 |       PX BLOCK ITERATOR    |          |     1 |     2 |  Q1,00 | PCWC |            |
|   7 |        TABLE ACCESS FULL   | T        |     1 |     2 |  Q1,00 | PCWP |            |
|   8 |     PX PARTITION HASH ALL  |          |     1 |     2 |  Q1,01 | PCWC |            |
|   9 |      TABLE ACCESS FULL     | T        |     1 |     2 |  Q1,01 | PCWP |            |
-----------------------------------------------------------------------------------------

   3 - access("T1"."ID"="T2"."ID")

SQL> SELECT dfo_number, tq_id, server_type, process, num_rows, bytes
  2  FROM v$pq_tqstat
  3  ORDER BY dfo_number, tq_id, server_type DESC, process;

DFO_NUMBER      TQ_ID SERVER_TYP PROCES   NUM_ROWS      BYTES
---------- ---------- ---------- ------ ---------- ----------
         1          0 Producer   P002        29042    3136278
         1          0 Producer   P003        70958    7673358
         1          0 Consumer   P000        20238    2188357
         1          0 Consumer   P001        79762    8621279
         1          1 Producer   P000        20238    4376714
         1          1 Producer   P001        79762   17242534
         1          1 Consumer   QC         100000   21619248
```

上面的输出给出了以下信息。

- 操作5通过从属进程P002发送了29 042条记录，而通过P003发送了70 958条记录。
- 操作4接收到由操作5发送的数据：通过从属进程P000发送的20 238条，以及通过从属进程P001发送的79 762条记录。这显示了在这个特定的案例中，基于分区键的分布运行得并不是很理想。
- 操作2通过从属进程P000发送给查询协调器20 238条记录，并通过从属进程P001发送了79 762条记录。作为上一次分布的结果，这一次依旧是不理想的。

15

❑ 操作1，也就是由查询优化器执行的操作，接收到100 000条记录。

每个从属进程都打开了它们自己到数据库实例之间的会话。这意味着如果想要监控或跟踪一条单独的SQL语句执行的处理过程，你没办法将注意力集中在单独的一个会话上。因此，你要么使用一个类似于实时监控这样的工具，以便为你整合来自多个会话的执行统计信息，要么就需要亲自动手做这件事。举例来说，通过SQL跟踪，每个从属进程都会生成其自己的跟踪文件（在这样的情形下TRCSESS命令行工具可能有所帮助）。与此有关的一个主要问题是因为当前实现的限制，查询协调器会忽略为其工作的从属进程的执行统计信息。下面由dbms_xplan生成的执行计划证实了这一点。注意除了由查询协调器执行的操作（PX COORDINATOR）以外，Starts、A-Rows以及Buffers这些列的值是都被设置为0：

```
SQL> SELECT * FROM table(dbms_xplan.display_cursor('6j5z013saaz9r',0,'iostats last'));

-----------------------------------------------------------------------------
| Id  | Operation                  | Name     | Starts | A-Rows | Buffers |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |          |      1 |   100K|      16 |
|   1 |  PX COORDINATOR            |          |      1 |   100K|      16 |
|   2 |   PX SEND QC (RANDOM)      | :TQ10001 |      0 |     0 |       0 |
|*  3 |    HASH JOIN               |          |      0 |     0 |       0 |
|   4 |     PX RECEIVE             |          |      0 |     0 |       0 |
|   5 |      PX SEND PARTITION (KEY)| :TQ10000 |      0 |     0 |       0 |
|   6 |       PX BLOCK ITERATOR    |          |      0 |     0 |       0 |
|*  7 |        TABLE ACCESS FULL   | T        |      0 |     0 |       0 |
|   8 |     PX PARTITION HASH ALL  |          |      0 |     0 |       0 |
|   9 |      TABLE ACCESS FULL     | T        |      0 |     0 |       0 |
-----------------------------------------------------------------------------
```

当处理库缓存中的游标时，一个可行的解决方案是不将format参数的值指定为last。事实上，如果SQL语句只执行了一次（可以通过查看PX COORDINATOR 操作的Starts列来检查），你会看见所有由从属进程执行的工作总和。遗憾的是，在使用了多个数据流操作的执行计划中，这个解决方案不会奏效，或者当处于一个RAC环境中时，从属进程的一部分是在一个或多个远程实例中分配的时候也不会有作用。下面的例子，显示了与上一个例子中相同的子游标，演示了它起作用的一个案例：

```
SQL> SELECT * FROM table(dbms_xplan.display_cursor('6j5z013saaz9r',0,'iostats'));

--------------------------------------------------------------------------------------
| Id  | Operation                  | Name     | Starts | A-Rows | Buffers | Reads  |
--------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |          |      1 |   100K|      16 |      0 |
|   1 |  PX COORDINATOR            |          |      1 |   100K|      16 |      0 |
|   2 |   PX SEND QC (RANDOM)      | :TQ10001 |      0 |     0 |       0 |      0 |
|*  3 |    HASH JOIN               |          |      2 |   100K|    2719 |   2464 |
|   4 |     PX RECEIVE             |          |      2 |   100K|       0 |      0 |
|   5 |      PX SEND PARTITION (KEY)| :TQ10000 |      0 |     0 |       0 |      0 |
|   6 |       PX BLOCK ITERATOR    |          |      2 |   100K|    2767 |   2464 |
|*  7 |        TABLE ACCESS FULL   | T        |     26 |   100K|    2767 |   2464 |
|   8 |      PX PARTITION HASH ALL |          |      2 |   100K|    2719 |   2464 |
|   9 |       TABLE ACCESS FULL    | T        |      2 |   100K|    2719 |   2464 |
--------------------------------------------------------------------------------------
```

执行并行DML语句的会话（而且仅对于那个会话而言；对于其他会话，未提交的数据甚至不可见）

在不提交（或回滚）事务的情况下无法访问修改的表。此时提交（或回滚）之前执行的SQL语句会引发一个ORA-12838: cannot read/modify an object after modifying it in parallel错误而终止。下面是一个例子（注意，UPDATE语句是并行化的）：

```
SQL> UPDATE t SET id = id + 1;

SQL> SELECT count(*) FROM t;
SELECT count(*) FROM t
                     *
ERROR at line 1:
ORA-12838: cannot read/modify an object after modifying it in parallel

SQL> COMMIT;

SQL> SELECT count(*) FROM t;

  COUNT(*)
----------
    100000
```

还有一个与上面这个限制类似的情况，当一个并行DML语句试图修改一个曾经被串行DML语句修改过的对象时，会引发一个ORA-12839: cannot modify an object in parallel after modifying it错误。下面是一个例子（注意SELECT FOR UPDATE语句，为了设置行锁，必须修改这张表）：

```
SQL> SELECT id FROM t WHERE rownum = 1 FOR UPDATE;

        ID
----------
      2343

SQL> UPDATE t SET id = id + 1;
UPDATE t SET id = id + 1
       *
ERROR at line 1:
ORA-12839: cannot modify an object in parallel after modifying it

SQL> COMMIT;

SQL> UPDATE t SET id = id + 1;

100000 rows updated.
```

15.4 直接路径插入

Oracle数据库提供两种将数据加载到表中的方法（假设该表不是存储在一个群集中）：传统插入和直接路径插入。传统插入，与其名称所示含义一样，就是通常使用的那一种。而数据库引擎只有在被明确要求的情况下才会使用直接路径插入。直接路径插入的目标是高效加载大量数据（对于小数据量，使用直接路径时的性能可能会比使用传统插入时更低）。直接路径插入能够实现高性能插入，是因为它的实现是以牺牲功能为代价来换取最优的性能。出于这个原因，与使用传统插入相比，使用直接路

径插入时会遇到更多的要求和限制。在本节中，我会讨论直接路径插入如何工作，什么时候适合使用这种技术，以及与之有关的一些陷阱和谬误。

注意　要加载数据CTAS语句，请使用直接路径插入。

15.4.1　工作原理

可以通过指定一个hint，或可以通过使用某个特定功能来启用直接路径插入。启用方式有以下几种可能性。

- 在INSERT INTO ... SELECT ...语句（包括多重插入）和MERGE语句（对于插入数据的部分）中指定append hint：

```
INSERT /*+ append */ INTO ... SELECT ...
```

- 在使用“普通”VALUES子句的INSERT语句中指定append hint（仅在11.1版本中有效）：

```
INSERT /*+ append */ INTO ... VALUES (...)
```

- 在使用“普通”VALUES子句的INSERT语句中指定append_values hint（仅从11.2版本开始有效）：

```
INSERT /*+ append_values */ INTO ... VALUES (...)
```

- 以并行方式执行INSERT INTO ... SELECT ...语句。注意在这种情况下，可以分别并行化INSERT和SELECT。要利用直接路径插入功有，至少INSERT部分必须要并行化。
- 直接使用OCI直接路径接口，或通过使用一个OCI直接路径接口的应用程序（例如，SQL*Loader实用工具）。

如果需要对一条自动启用了该功能的SQL语句禁用直接路径插入（例如，一条并行执行的INSERT INTO ... SELECT ...语句），可以指定noappend这个hint。

为改善效率，直接路径插入会直接在被修改段的高水位线以上使用直接路径写来加载数据，通过这种方式来提高性能。这个事实的存在有重要的意义。

- 缓冲区缓存，因为直接写的原因，被绕过了。
- 并发的DELETE、INSERT、MERGE以及UPDATE语句，与在修改的段上建立（或重建）的索引一样，是不被允许的。当然，为保证这一点，段锁（segment lock）会被获取。
- 在高水位线以下包含的空闲空间块不在考虑范围之内。这意味着即使为了清除数据而执行了DELETE语句，段的大小还是会不断地增大。

直接路径插入能够带来更好性能的一个原因是，对于表段来说只会生成最少量的undo。事实上，只有空间管理操作才会少量生成undo（例如，为了增加高水位线以及向段中增加新的内容），而对于那些通过直接路径插入存储到数据块中的记录来说，则不会生成undo。但是，如果表上有索引，对于索引段来说一般又会生成undo。如果你还想要避免与索引段相关的undo，可以在加载之前禁用索引并在加载完毕后重建索引。尤其是在ETL任务中，这是实践中常用的操作。而且大家喜欢这样做的另一个原因是，让数据库引擎在加载结束时自行维护索引可能不如重建索引来得更快。

为进一步改进性能，还可以使用最小化日志（minimal logging）。最小化日志的目的是最小化redo

的生成。这个操作是可选的，但是通常它对于大幅减少响应时间的作用十分明显。可以通过在表或分区级别设置nologging参数来指示数据库引擎使用最小化日志。一定要理解最小化日志仅支持直接路径插入和一部分DDL语句。事实上，redo总是会为所有操作生成。要知道对于存储在群集中的表，最小化日志无能为力。

注意 只有在已经完全理解指定nologging，也就是最小化redo生成的影响的时候，你才可以这样做。事实上，对于使用最小化日志修改的块，无法执行介质恢复。这意味着如果执行了介质恢复，数据库引擎只能将那些使用了nologging的块标记为逻辑损坏，因为介质恢复需要访问redo信息以便于重构块的内容，而这对于nologging的块是不可能的，因为之前提到使用最小化日志时redo信息是不会存储下来的。因此，访问包含这些块的对象的SQL语句会引发一个ORA-26040: Data block was loaded using the NOLOGGING option错误而终止。因此，应该仅在以下情况下使用最小化日志：能够手动重新加载数据，或愿意在加载完毕后执行一个备份，或者可以承担丢失数据的风险。

图15-11展示了一个你可以通过直接路径插入实现的改进的例子。这些数据是我在测试系统上通过启动dpi_performance.sql这个脚本测量出来的。

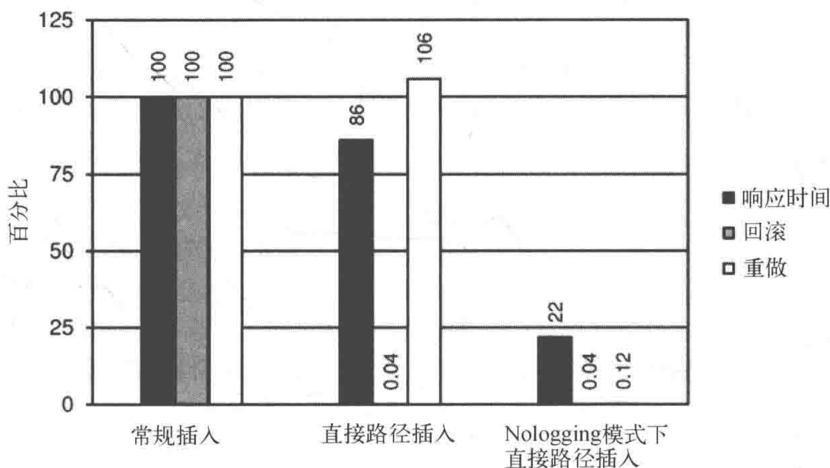


图15-11 在使用和不使用直接路径插入的情况下加载数据的对比（没有索引的表）

注意，在图15-11中，对于两种直接路径插入，undo的生成都是可以忽略不计的。这是因为被修改的表没有索引。图15-12展示了同样的测试在适当的位置使用了主键时的数据。不出所料，为索引段生成了undo。

直接路径插入不像传统插入那样支持所有对象。它们的功能是受限的。如果数据库引擎无法执行直接路径插入，则该操作默认会转化成为一个传统插入。当遇到下列条件之一的时候就会发生这种情况。

- 已启用的INSERT触发器出现在修改的表上。（注意，DELETE和UPDATE触发器对直接路径插入没有影响）

- 已启用的外键出现在要修改的表上（指向修改表的其他表的外键没有问题）。
- 修改的表是索引组织表。
- 修改的表存储在群集中。
- 修改的表包含对象类型的列。
- 修改的表拥有一个通过非唯一索引维护的主（或唯一）键。从11.1版本开始，这个限制不复存在。

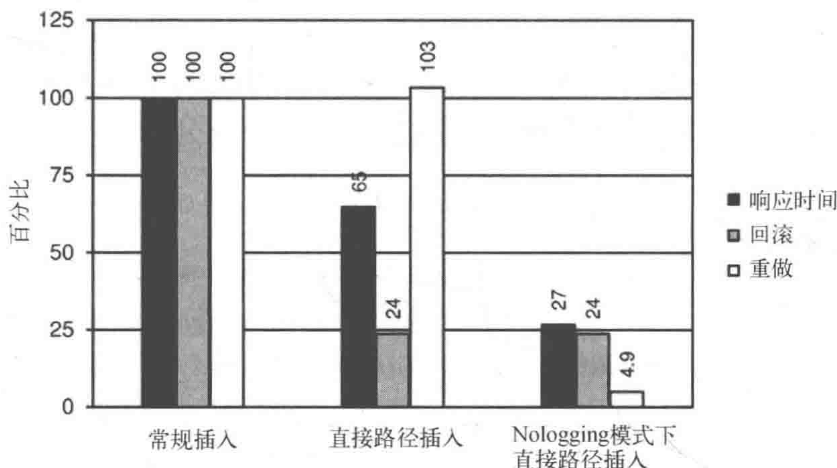


图15-12 在使用和不使用直接路径插入的情况下加载数据的对比（带有主键的表）

15.4.2 何时使用

一旦需要加载大量的数据，而且适用于直接路径插入的那些限制条件对你来说不是问题的时候，就应该使用直接路径插入。

如果提高性能是你的主要目的，你可能还需要考虑使用最小化日志（nologging）。然而，正如之前解释过的，只有在完全理解并接受这样做的影响，并且你能够采取必要的措施来保证不在处理过程中损失数据的情况下，才可以使用此选项。

15.4.3 陷阱和谬误

即使没有使用最小化日志，在noarchivelog模式下运行的数据库也不会为直接路径插入生成redo。

如果数据库或表空间处于force logging模式下，对存储在其中的段使用最小化日志是不可能的。事实上，force logging会覆盖nologging参数。注意当使用类似或Streams这样的主从复制特性的时候，force logging尤其有用。为了能够成功运用这些技术，重做日志中需要包含关于所有数据修改的信息。

在直接路径插入期间，高水位线并不增长。只有提交事务的时候才会执行增长的操作。因此，执行直接路径插入的会话（而且仅对于此会话；对于其他的会话，高水位线以上的未提交数据甚至不可见），在加载完毕后在不提交（或回滚）事务的情况下无法访问修改的表。在提交（或回滚）之前执行的SQL语句会伴随着一个ORA-12838: cannot read/modify an object after modifying it in parallel

错误而终止。下面是一个例子：

```
SQL> INSERT /*+ append */ INTO t SELECT * FROM t;

SQL> SELECT count(*) FROM t;
SELECT count(*) FROM t
                     *
ERROR at line 1:
ORA-12838: cannot read/modify an object after modifying it in parallel

SQL> COMMIT;

SQL> SELECT count(*) FROM t;

  COUNT(*)
----------
     10000
```

与0RA-12938错误有关的文本可能会令人迷惑，因为即使没有使用并行处理也会这样生成。

15.5 行预取

当一个应用程序从数据库中提取数据，它可以逐行提取，或者使用更好的方式，同时提取很多行。同时提取很多行称作行预取（row prefetching）。

15.5.1 工作原理

行预取的概念简单明确。应用程序每次请求驱动程序从数据库中检索一行数据，都有多出来的行通过行预取被预取出来，存放在客户端内存中。这种方式下，后续几个请求不需要再次执行数据库访问来提取数据。检索数据可以由客户端内存直接提供。因此，往返数据库的次数随着预取行的数量成比例下降。所以，检索包含很多行数据的结果集时的负载可能会大为减少。作为一个例子，图15-13向你展示在将预提取的行数增加至50的时候，检索100 000行数据的响应时间。这个测试使用了`RowPrefetchingPerf.java`文件中的Java类。

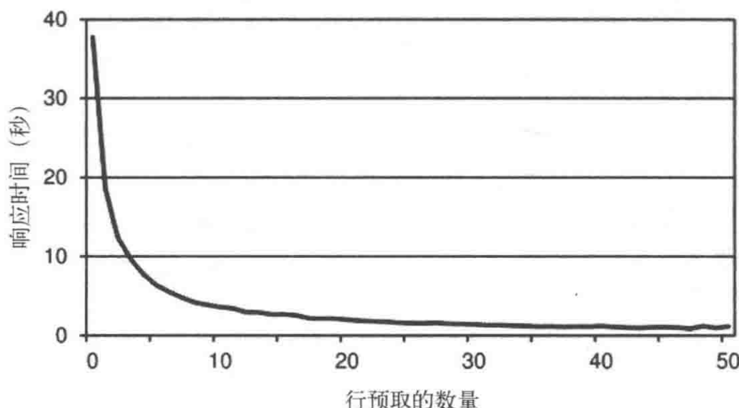


图15-13　检索包含很多行数据的结果集，所需的时间强烈依赖于预提取行的数量

一定要理解在不使用行预取时（也就是说，逐行处理），检索的不良性能表现并非是由数据库引擎引起的。相反，这是由应用程序自己引起的，并因此承担后果。在查看为非预取的案例使用SQL跟踪生成的执行统计信息后，原因就更明显了。下面的执行统计信息显示，虽然客户端花费了大概持续37秒钟的时间（参见图15-13），但其中只有2.3秒是花费在处理数据库端的查询上！

```
call     count       cpu    elapsed       disk      query    current       rows
------- ------  -------- ---------- ---------- ---------- ---------- ----------
Parse        1      0.00       0.00          0          0          0          0
Execute      1      0.00       0.00          0          0          0          0
Fetch   100001      2.14       2.30        213     100004          0     100000
------- ------  -------- ---------- ---------- ---------- ---------- ----------
total   100003      2.14       2.30        213     100004          0     100000
```

即便行预取对于客户端来讲更加重要，但数据库也能从中获益。事实上，行预取极大地减少了逻辑读的数量（从100 004下降到3542）。下面的执行统计信息显示当预取50行的时候减少的逻辑读数量：

```
call     count       cpu    elapsed       disk      query    current       rows
------- ------  -------- ---------- ---------- ---------- ---------- ----------
Parse        1      0.00       0.08          0          0          0          0
Execute      1      0.00       0.00          0          0          0          0
Fetch     2001      0.11       0.13        665       3542          0     100000
------- ------  -------- ---------- ---------- ---------- ---------- ----------
total     2003      0.11       0.21        665       3542          0     100000
```

接下来的小节提供一些关于如何在使用PL/SQL、OCI、JDBC、ODP.NET以及PHP时利用行预取的基础知识。除了由每个API提供的功能以外，从12.1版本开始，有一个应用程序设置的值可以被Oracle客户端目录下的$TNS_ADMIN/oraaccess.xml配置文件覆盖。注意因为它是一个客户端配置文件，所以PL/SQL引擎不受它的影响。但不管怎样，通过OCI库连接的所有应用程序都会受其影响。下面的例子展示如何为所有的连接设置行预取值为100：

```
<?xml version="1.0" encoding="ASCII" ?>
  <oraaccess xmlns="http://xmlns.oracle.com/oci/oraaccess"
             xmlns:oci="http://xmlns.oracle.com/oci/oraaccess"
             schemaLocation="http://xmlns.oracle.com/oci/oraaccess
                             http://xmlns.oracle.com/oci/oraaccess.xsd">
  <default_parameters>
    <prefetch>
      <rows>100</rows>
    </prefetch>
  </default_parameters>
</oraaccess>
```

注意　要利用$TNS_ADMIN/oraaccess.xml配置文件，只要求客户端可执行文件必须是12.1版本的。换句话说，数据库版本无关紧要。

关于$TNS_ADMIN/oraaccess.xml配置文件的详细信息，请参考*Oracle Call Interface Programmer's Guide*手册。

1. PL/SQL

如果在编译时将plsql_optimize_level初始化参数设置为2（默认值）或更高，则行预取会用于游标FOR循环。举个例子，下面PL/SQL代码块中的查询每次预提取100行数据：

```
BEGIN
  FOR c IN (SELECT * FROM t)
  LOOP
    -- process data
    NULL;
  END LOOP;
END;
```

注意 预提取的行数无法进行更改。

一定要记住行预取仅会自动用于FOR循环游标。要在其他类型的游标中使用行预取，必须使用BULK COLLECT子句。此处展示它在一个隐式游标中的使用方式：

```
DECLARE
  TYPE t_t IS TABLE OF t%ROWTYPE;
  l_t t_t;
BEGIN
  SELECT * BULK COLLECT INTO l_t
  FROM t;
  FOR i IN l_t.FIRST..l_t.LAST
  LOOP
    -- process data
    NULL;
  END LOOP;
END;
```

通过上面的PL/SQL代码块，会在单独的一次提取中返回结果集的所有行。如果行的数量很多，会需要大量的内存。因此，在实践中，除非你知道即将要返回的行数量是做过限制的，否则就应该使用LIMIT子句为单独的一次提取设置一个限制。下面的PL/SQL代码块展示如何一次提取100行数据：

```
DECLARE
  CURSOR c IS SELECT * FROM t;
  TYPE t_t IS TABLE OF t%ROWTYPE;
  l_t t_t;
BEGIN
  OPEN c;
  LOOP
    FETCH c BULK COLLECT INTO l_t LIMIT 100;
    EXIT WHEN l_t.COUNT = 0;
    FOR i IN l_t.FIRST..l_t.LAST
    LOOP
      -- process data
      NULL;
    END LOOP;
  END LOOP;
  CLOSE c;
END;
```

dbms_sql包、本地动态SQL以及RETURNING子句都支持行预取。然而，如在之前的两个例子中所示，必须显式启用（例如，使用BULK COLLECT）行预取。

2. OCI

使用OCI，行预取是由两个属性控制的：OCI_ATTR_PREFETCH_ROWS和OCI_ATTR_PREFETCH_MEMORY。前者限制提取的行数量。后者限制用于提取行的内存总量（按字节计）。下面的代码片段展示如何调用OCIAttrSet函数来设置这些属性。完整的例子由row_prefetching.c文件中的C程序提供：

```
ub4 rows = 100;
OCIAttrSet(stm,                    // 语句句柄
           OCI_HTYPE_STMT,         // 被修改的句柄类型
           &rows,                  // 特性的值
           sizeof(rows),           // 特性的值的大小
           OCI_ATTR_PREFETCH_ROWS, // 要设置的特性
           err);                   // 错误句柄

ub4 memory = 10240;
OCIAttrSet(stm,                      // 语句句柄
           OCI_HTYPE_STMT,           // 被修改的句柄类型
           &memory,                  // 特性的值
           sizeof(memory),           // 特性的值的大小
           OCI_ATTR_PREFETCH_MEMORY, // 要设置的特性
           err);                     // 错误句柄
```

同时设置两个属性时，首先达到的那个限制会被执行。要关掉行预取，必须将两个属性都设置为0。

3. JDBC

Oracle JDBC驱动程序默认情况下会启用行预取。你可以通过两种方式变更提取行的默认数量（10）。第一种是当通过OracleDataSource或OracleDriver类打开一个到数据库引擎的连接时指定一个属性。下面的代码片段作为例子展示如何为一个OracleDataSource对象设置用户名、密码，以及预提取的行数量。注意，在本例中，因为它被设置为1，所以行预取被禁用了：

```
connectionProperties = new Properties();
connectionProperties.put(OracleConnection.CONNECTION_PROPERTY_USER_NAME, user);
connectionProperties.put(OracleConnection.CONNECTION_PROPERTY_PASSWORD, password);
connectionProperties.put(OracleConnection.CONNECTION_PROPERTY_DEFAULT_ROW_PREFETCH, "1");
dataSource.setConnectionProperties(connectionProperties);
```

第二种方式是在连接级别通过使用java.sql.Statement或java.sql.ResultSet接口（以及它们的子接口）的 setFetchSize方法来覆盖默认值。下面的代码片段展示使用setFetchSize方法将提取的行数量设置为100的例子。RowPrefetching.java文件中的Java程序提供了完整的例子：

```
sql = "SELECT id, pad FROM t";
statement = connection.prepareStatement(sql);
statement.setFetchSize(100);
resultset = statement.executeQuery();
while (resultset.next())
{
  id = resultset.getLong("id");
  pad = resultset.getString("pad");
  // 过程数据
```

```
}
resultset.close();
statement.close();
```

4. ODP.NET

ODP.NET的默认提取大小（65 536）是按字节定义的，而不是按行定义。可以通过OracleCommand和OracleDataReader类提供的FetchSize属性来更改这个值。下面的代码片段是如何设置该属性的值以达到提取100行的一个例子。注意如何使用OracleCommand 类的RowSize属性计算存储100行数据所需的内存总量。RowPrefetching.cs文件中的C#程序提供一个完整的例子：

```
sql = "SELECT id, pad FROM t";
command = new OracleCommand(sql, connection);
reader = command.ExecuteReader();
reader.FetchSize = command.RowSize * 100;
while (reader.Read())
{
  id = reader.GetDecimal(0);
  pad = reader.GetString(1);
  // 过程数据
}
reader.Close();
```

从10.2.0.3版本的ODP.NET开始，也可以通过下面的注册表条目来更改默认提取大小（<Assembly_Version>是Oracle.DataAccess.dll的完整版本号）：

```
HKEY_LOCAL_MACHINE\SOFTWARE\ORACLE\ODP.NET\<Assembly_Version>\FetchSize
```

5. PHP

在PECL OCI8扩展程序中默认是启用行预取的。可以通过两种方式更改默认的提取行的数量（100；直到该扩展的1.3.3版本，它一直是10）。第一种是通过设置php.ini配置文件中的oci8.default_prefetch选项来更改默认值。第二种是在语句级别通过在解析和执行阶段之间调用oci_set_prefetch函数来覆盖默认值。下面的代码片段是一个如何设置此值以提取100行数据的例子。RowPrefetching.php脚本提供一个完整的例子：

```
$sql = "SELECT id, pad FROM t";
$statement = oci_parse($connection, $sql);
oci_set_prefetch($statement, 100);
oci_execute($statement, OCI_NO_AUTO_COMMIT);
while ($row = oci_fetch_assoc($statement))
{
  $id = $row['ID'];
  $pad = $row['PAD'];
  // 过程数据
}
oci_free_statement($statement);
```

15.5.2 何时使用

不管怎样，当需要提取的数据超过一行的时候，使用行预取就是合理的。

15.5.3 陷阱和谬误

当使用OCI库的时候，并非总是能够完全禁用行预取。举个例子，使用JDBC OCI驱动程序或使用SQL*Plus，提取行数量的最小值是2。在实践中，这不会成为问题，至于为什么可能还会有一些疑惑，例如，尽管在SQL*Plus中将arraysize系统变量设置为1，你还是会看到有两行数据被提取了。

比如说，如果一个应用程序一次显示10行数据，一般来说从数据库中提取100行是没有意义的。预提取的行数应该尽可能与应用程序在特定的时间点需要的行数相匹配。

15.6 数组接口

前面的章节展示了当一个应用程序从数据库中提取数据的时候，它可以逐行提取，或者使用更好的做法通过行预取提取多行。同样的理念也适用于应用程序向数据库引擎发送数据时的情况，或者换句话说，在输入变量的绑定期间。此时，**数组接口**（array interface）就可以派上用场了。

15.6.1 工作原理

使用数据接口可以绑定数组而非标量值。当某个特定的DML语句需要插入或修改大量数据时，这个特性尤其有用。不用为每一行记录单独执行该DML语句，你可以将所有必要的值绑定为一个数组，并且仅需要执行一次，或者如果行的数量很大，可以将执行拆分为多个小一些的批次。这样，到数据库的往返次数就会随着数组的大小成比例的减少。图15-14展示通过将数组的大小提高到50的时候插入100 000行数据的响应时间。此测试使用了ArrayInterfacePerf.java文件中的Java类。

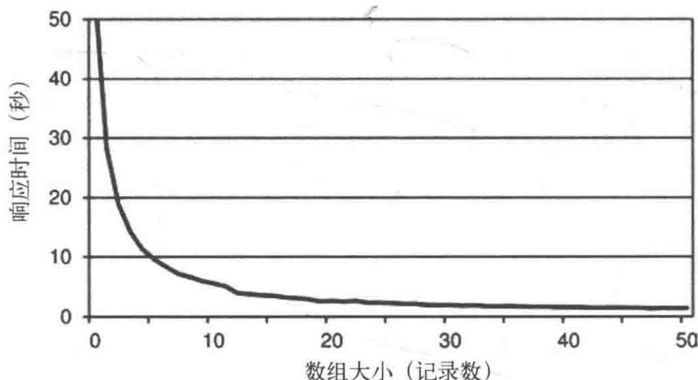


图15-14　向数据库加载数据所需的时间强烈依赖于每次执行所处理的行数

一定要理解在不使用数组处理（也就是逐行处理）的情况下，加载的不良性能表现并非在于数据库引擎。相反，这是由应用程序自身引起的，并因而承担后果。通过查看使用SQL跟踪生成的执行统计信息，可以很明显地发现这一点。下面的执行统计信息显示，虽然客户端花费了持续超过50秒的时间（见图15-14），但其中只有3.1秒是花费在数据库端对插入的处理上：

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------ -------- ---------- ---------- ---------- ---------- ----------
Parse        1     0.00       0.00          0          0          0          0
```

```
Execute 100000     3.06       3.10          2       2075     114173     100000
Fetch        0     0.00       0.00          0          0          0          0
------- ------ -------- ---------- ---------- ---------- ---------- ----------
total   100001     3.06       3.10          2       2075     114173     100000
```

尽管数组接口对于客户端来说更加高效，但是数据库引擎也同样从中获益。事实上，数组接口减少了逻辑读的数量（从116 248下降到18 143）。下面的执行统计信息显示以50为批次插入数据时减少的逻辑读数量：

```
call     count      cpu    elapsed       disk      query    current       rows
------- ------ -------- ---------- ---------- ---------- ---------- ----------
Parse        1     0.00       0.00          0          0          0          0
Execute   2000     0.26       0.38          0       3132      15011     100000
Fetch        0     0.00       0.00          0          0          0          0
------- ------ -------- ---------- ---------- ---------- ---------- ----------
total     2001     0.26       0.38          0       3132      15011     100000
```

下面的小节提供一些关于如何在使用PL/SQL、OCI、JDBC以及ODP.NET时利用数组接口的基础知识。注意，在PHP中，对于PECL OCI8扩展程序，是不支持数组接口的。但根据使用PHP开发的应用程序类型，我认为这不是一个大问题。

1. PL/SQL

要在PL/SQL中使用数组接口，可以使用FORALL语句。通过它，可以执行一条使用绑定数组向数据库引擎传递数据的DML语句。下面的PL/SQL代码块展示如何在单独的一次执行中插入100 000行数据。注意，代码的第一部分仅用于准备数组。带有INSERT语句的FORALL语句本身仅占用了PL/SQL代码块中的最后两行：

```
DECLARE
  TYPE t_id IS TABLE OF t.id%TYPE;
  TYPE t_pad IS TABLE OF t.pad%TYPE;
  l_id t_id := t_id();
  l_pad t_pad := t_pad();
BEGIN
  -- prepare data
  l_id.extend(100000);
  l_pad.extend(100000);
  FOR i IN 1..100000
  LOOP
    l_id(i) := i;
    l_pad(i) := rpad('*',100,'*');
  END LOOP;
  -- insert data
  FORALL i IN l_id.FIRST..l_id.LAST
    INSERT INTO t VALUES (l_id(i), l_pad(i));
END;
```

一定要注意，即使该语法是基于FORALL关键字的，这也并不是一个循环。所有的数据行是在单独的一次数据库调用中发送的。

数组接口受支持的情况不止此一种，还有dbms_sql包以及本地动态SQL也支持它。

2. OCI

要通过OCI利用数组接口，不需要具体的函数。事实上，用于绑定变量的函数OCIBindByPos和OCIBindByName，以及用于执行SQL语句的函数OCIStmtExecute，都可以使用数组作为参数。array_interface.c文件中的C程序提供了一个例子。

3. JDBC

要通过JDBC利用数组接口，可以使用批量更新。如下面的代码片段所示，在单独的一次执行中插入100 000行数据，可以通过执行addBatch方法将一次“执行”添加到一个批次中。当包含多个“执行”的整个批次准备就绪，可以通过执行executeBatch方法向数据库引擎提交该批次数据。两个方法都在java.sql.Statement接口中提供，而且因此，也在子接口java.sql.PreparedStatement和java.sql.CallableStatement中提供。完整的例子由ArrayInterface.java文件中的Java程序提供：

```
sql = "INSERT INTO t VALUES (?, ?)";
statement = connection.prepareStatement(sql);
for (int i=1 ; i<=100000 ; i++)
{
  statement.setInt(1, i);
  statement.setString(2, "... some text ...");
  statement.addBatch();
}
counts = statement.executeBatch();
statement.close();
```

警告　根据JDBC标准，java.sql.Statement接口以及它的子接口java.sql.PreparedStatement和java.sql.CallableStatement都支持批量更新。尽管Oracle的实现支持标准的API，可以预期的是，仅当使用java.sql.PreparedStatement接口重复执行拥有不同绑定变量的同一条SQL语句时才会有性能提升。

---

4. ODP.NET

要通过ODP.NET使用数组接口，基于数组定义参数，并将存储在数组中值的数量设置为ArrayBindCount属性的值就可以了。下面的代码片段，通过在单独的一次执行中插入100 000行数据，证实了这一点。可以在ArrayInterface.cs文件中的C#程序中找到完整的例子：

```
Decimal[] idValues = new Decimal[100000];
String[] padValues = new String[100000];

for (int i=0 ; i<100000 ; i++)
{
  idValues[i] = i;
  padValues[i] = "... some text ...";
}

id = new OracleParameter();
id.OracleDbType = OracleDbType.Decimal;
id.Value = idValues;
```

```
pad = new OracleParameter();
pad.OracleDbType = OracleDbType.Varchar2;
pad.Value = padValues;

sql = "INSERT INTO t VALUES (:id, :pad)";
command = new OracleCommand(sql, connection);
command.ArrayBindCount = idValues.Length;
command.Parameters.Add(id);
command.Parameters.Add(pad);
command.ExecuteNonQuery();
```

15.6.2 何时使用

无论何时需要插入或修改超过一行的数据时，使用数组接口就是合理的。你只需要考虑在客户端可能会因为存储数组而需要更多的内存。通常，这都不会成为问题，除非使用的数组大小很夸张。

15.6.3 陷阱和谬误

在通过SQL跟踪生成的执行统计中，没有明显的关于使用数组处理的信息。但是，如果你知道SQL语句是哪一个，通过查看修改的行数和执行的次数之间的比率，应该能够确定是否使用了数组处理。举例来说，在下面的执行统计中，一个普通的INSERT语句，只被执行了一次，插入了2342行数据。这样的结果可能只会在使用数组接口时才会出现：

```
INSERT INTO T VALUES (:B1 , :B2 )

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ---------- ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0         78        522        2342
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ---------- ----------
total        2      0.00       0.00          0         78        522        2342
```

15.7 小结

本章描述了几种致力于改进性能的高级优化技术。其中的一些优化技术（物化视图、结果缓存、并行处理和直接路径插入），只应该在“正常”的优化技术无法实现要求的性能时才去使用。比较起来，其他的优化技术（行预取和数组处理）则应该尽可能多地使用。

尽管本章主要描述那些并不常用的优化技术，但是接下来（最后）的一章涵盖的优化技术，基本上适用于在数据库中存储的每一张表。事实上，当你执行从逻辑设计到物理设计的转变的时候，有必要确定每一张表在物理上是如何存储数据的。

第 16 章

优化物理设计

在从逻辑设计向物理设计的转换过程中，必须做出四种类型的决策。第一，对于每一张表，不仅要决定是否应该使用堆表、群集或者索引组织表，而且还要决定是否需要使用分区。第二，必须考虑是否应该利用诸如索引或物化视图等冗余访问结构。第三，必须决定如何实现约束（这里不是讨论你是否必须实现它们）。第四，必须决定数据如何在块中存储，包括列的顺序，使用什么样的数据类型，每个块中应该存储多少行数据，或者是否应该激活压缩功能。本章只关注第四个主题。关于其他三个主题的信息，尤其是前两个，请参考第13章、第14章和第15章。

本章的目标是解释为何不应该将物理设计的优化视为微调的活动，而是作为一项基本的优化技术。本章的起点是讨论为何选择正确的列顺序和正确的数据类型事关重大。接下来会解释什么是行迁移和行链接，如何定位与它们有关的问题，以及如何从一开始就避免行迁移和行链接。然后，本章会描述拥有高负载的系统会经历的一个常见性能问题：块争用。最后，本章还会描述如何利用数据压缩来改进性能。

16.1　最优列顺序

我们通常很少会将注意力放在如何为一张表找出最优列顺序上面。根据具体情况，列顺序既可能没有丝毫影响，也可能会引发显著的开销。要理解什么情况下可能会引发显著的开销，就十分有必要讲述一下数据库引擎是如何在块中存储数据的。

在块中存储一行数据有着非常简单的格式（见图16-1）。首先，有一个头部（H）记录着关于数据行本身的一些属性，比如它是否被锁定或它包含了多少个列。然后，是各个列。因为每个列都可能拥有不同的大小，所以它们中的每一个都由两部分组成。第一部分是数据的长度（L*n*）。第二部分是数据本身（D*n*）。

H	L1	D1	L2	D2	L3	D3		L*n*	D*n*

图16-1　数据块中存储的数据行的格式（H =行的头部，L*n* = 第*n*列的长度，D*n* = 第*n*列的数据）

在这个格式中要理解的重点是数据库引擎不知道一行数据中各个列的偏移量。举例来说，如果它必须要定位列3，那么它不得不从定位列1开始（这个简单，因为头部的长度是已知的）。然后，根据列1的长度，它定位到列2。最后，根据列2的长度，它定位到列3。所以无论何时当行里面包含很多列，

那么定位接近开头位置的列要比定位接近行末尾的列要快得多。为了更好地理解此内容，可以执行下面的测试，该测试来自column_order.sql脚本，用来测量与列的搜索有关的开销。

(1) 创建一张拥有250个列的表：

```
CREATE TABLE t (n1 NUMBER, n2 NUMBER, ..., n249 NUMBER, n250 NUMBER)
```

(2) 插入10 000条数据。每一行的每一列都存储相同的值。

(3) 为下面的查询测量响应时间，为每个列循环执行1000次：

```
SELECT count(<col>) FROM t
```

图16-2总结了这个测试在我的测试服务器上运行的结果。需要注意的是，引用第一个列（位置1）的查询执行速度是引用第250个列（位置250）的查询的五倍。这是因为数据库引擎优化了每次访问，而且因此避免了多余的列定位和读取的处理工作。举例来说，SELECT count(n3) FROM t这个查询在定位到第三个列以后，就停止了后续检索数据行的操作。图16-2同样报告了，在位置0，count(*)的计算，根本不需要访问任何列。

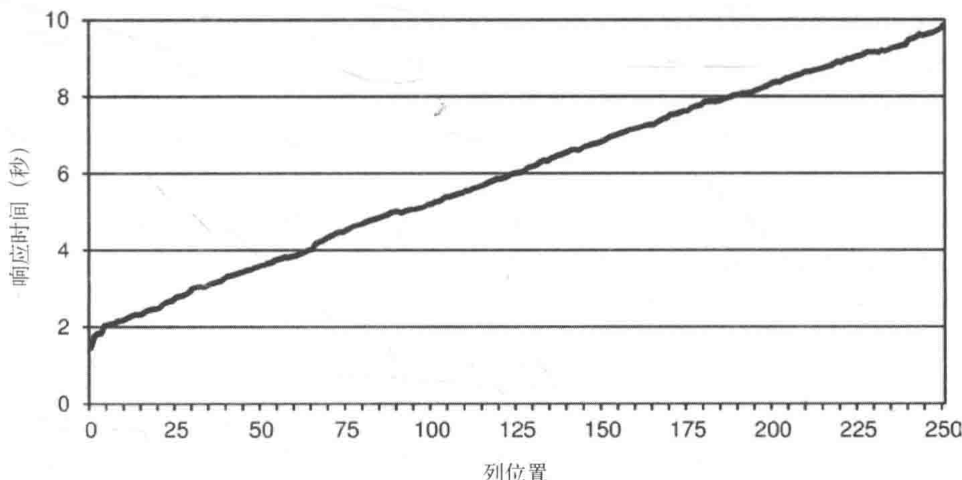


图16-2　一个列在数据行中的位置与访问它需要的处理总量

因为如此，惯用规则是首先放置需要经常访问的列。然而，为了利用这种特性，你应该要注意只访问那些真正需要的列。无论如何，从性能的角度来看，查询不需要的列（或更糟糕的情况是，经常使用SELECT *引用所有的列，即使是只有一部分列是应用程序需要的）都是不好的，不仅因为从数据块中读取它们时存在着开销，而且就像你刚刚看到的那样，也是因为在服务器以及客户端上临时存储它们时需要更多的内存，而且在网络上发送它们也需要更多的时间和资源。简而言之，每次处理数据，都会有开销。

在实践中，与列的位置有关的开销在下列的情形当中是（更加）显著的。

- 当表拥有许多列，而且SQL语句经常引用存储于行末的列的很少一部分时。
- 当从一个块中读取很多行时，比如在全表扫描期间。这是因为，通常访问每个块中的少数行时，定位和访问一个块的开销远远高于仅读取少数行时定位和访问列的开销。举例来说，如果通过将PCTFREE设置为90（因此我降低了每个块中的记录数）来运行column_order.sql脚本，

16

引用第一个列的查询执行速度只比引用第250列的查询快不到两倍（如在图16-2中看到的，与将PCTFREE设置为10时相比，快了大约五倍）。

因为尾部的NULL值不存储，所以将预期会包含NULL值的列放置在表的末尾显得比较合理。通过这种方式，物理存储的列数量以及关联的平均行大小都有可能随之降低。

16.2 最优数据类型

最近这些年我见证了在物理设计方面的一个令人担忧的趋势，我称之为错误的数据类型选择（wrong datatype selection），我在1.2节中简单介绍过这一趋势。乍看之下，为一个列选择数据类型看起来像是要做出的一个非常直截了当的决定。然而，在这样的一个世界里，软件模式师通常会花费大量的时间来讨论诸如敏捷软件开发、SOA或持久层框架这样高层次的事情，大多数人看起来是忘记了低层次的事情。我相信十分有必要回归底层基础，并讨论为什么数据类型选择如此重要。

16.2.1 数据类型选择中的陷阱

为了演示数据类型选择中的错误，接下来我会展现曾经反复遇到过的五个典型问题的例子。

第一个由数据类型选择错误引起的问题，是在数据库中插入或修改数据时使用了错误的数据验证，或缺少数据验证。举例来说，如果一个列本来应该存储数字值，实际为它选择一个字符串数据类型，那么就需要一个外部校验。换句话说，数据库引擎无法校验数据。数据库引擎将这个工作留给应用程序去做。即使这样的一个验证很容易实现，也要牢记相比较集中在数据库而言，每次当这段相同的代码被分发到多个位置时，早晚会出现功能上不一致的情况（典型的例子，在某些位置上的校验可能被忘记了，或可能后来校验规则发生了改变，而实现规则的程序只在一部分位置上进行了更新）。我即将呈现的例子与nls_numeric_characters初始化参数有关。记住这个初始化参数指定的是小数和数值分隔符使用的特性。例如，在瑞士它经常被设置为“.,”，因此π的值被格式化成这样：3.14159。相反，在德国它通常被设置为“,.”，因此同样的一个值会被格式化为：3,14159。迟早，因为在数据库中使用了错误的数据类型，对该初始化参数使用了不同的客户端设置的应用程序，如果执行从VARCHAR2向NUMBER类型的转换，将会引发一个ORA-01722:invalid number错误。而且等到你注意到这个问题的时候，你的数据库将会被包含两种格式的VARCHAR2列填满，而且那时候就会需要执行令人头疼的数据校正。

第二个由数据类型选择错误引起的问题是信息的丢失。换句话说，在从原始的（正确的）数据类型向数据库的数据类型转换期间，信息会发生丢失。举例来说，想象一下当使用DATE数据类型存储一个事件的日期和时间，而不是使用TIMESTAMP WITH TIME ZONE数据类型时会发生什么。小数部分的秒和时区信息会丢失。尽管小数部分的秒导致的问题可能会被认为是小错误（不到1秒钟），而时区则可能是一个更大的问题。在我曾经亲身经历的一个案例中，一个客户的数据总是使用本地标准时间（没有夏令时调整）生成，并直接存储在数据库中。当出于报表的原因而必须应用一个夏令时修正的时候，问题就发生了。一个设计用于在两个时区之间进行转换的函数被实现出来。它的使用方法如下：

```
new_time_dst(in_date DATE, tz1 VARCHAR2, tz2 VARCHAR2) RETURN DATE
```

调用一个这样的函数非常快速。问题是在每个报表中都会成千上万次调用它。结果响应时间增加

了25倍。很明显，使用正确的数据类型，所有的事情不仅会更快，而且会更容易（转换会被自动执行）。

第三个由数据类型选择错误引起的问题是事情不像期望的那样运转。比如说你必须对一张表进行范围分区，基于一个存储着日期和时间信息的DATE或TIMESTAMP列。这通常没什么大不了的。如果分区键使用的列包含基于某种格式掩码的日期时间值的数字表现形式，或等价的字符串表现形式，代替了原本的DATE或TIMESTAMP值，此时就会发生问题。如果从日期时间值向数字值的转换是通过类似YYYYMMDDHH24MISS的格式掩码执行的，范围分区的定义仍然是有可能的。然而，如果转换是基于类似DDMMYYYYHH24MISS这样的格式掩码，因为数字（或字符串）顺序并非按自然的日期时间值顺序保存的，在不变更列的数据类型或格式的情况下你根本没有机会解决这个问题（自11.1版本开始，在某些情况下有可能通过实现基于虚拟列的分区解决这个问题）。

第四个由数据类型选择错误引起的问题与查询优化器有关。这可能是这个候选名单上最不明显的一个，也是导致问题时最微妙的一个。这个问题的原因是使用错误的数据类型，查询优化器会执行错误的估算，因此，选择的访问路径不是最优的。通常，像这样的事情发生的时候，大多数人会责怪查询优化器“又一次”没有做好本职工作。实际上，问题是你向查询优化器隐藏了信息，所以它无法正确地完成它的工作。为了更好地理解这个问题，看一下下面的例子，它来自wrong_datatype.sql这个脚本。在这里，你会看到对于类似的限制条件，三个存储相同数据集（2014年的每一天的日期）但是使用不同数据类型的列，估算出来的基数之间的不同。正如你所见到的，查询优化器只能够为正确定义的列做出合理的估算（正确的基数是28）：

```
SQL> CREATE TABLE t (d DATE, n NUMBER(8), c VARCHAR2(8));

SQL> INSERT INTO t (d)
  2  SELECT to_date('20140101','YYYYMMDD')+level-1
  3  FROM dual
  4  CONNECT BY level <= 365;

SQL> UPDATE t SET n = to_number(to_char(d,'YYYYMMDD')), c = to_char(d,'YYYYMMDD');

SQL> execute dbms_stats.gather_table_stats(ownname=>user, tabname=>'t')

SQL> SELECT * FROM t ORDER BY d;

D                  N C
--------- ---------- --------
01-JAN-14   20140101 20140101
02-JAN-14   20140102 20140102
...
30-DEC-14   20141230 20141230
31-DEC-14   20141231 20141231

SQL> EXPLAIN PLAN SET STATEMENT_ID = 'd' FOR
  2  SELECT *
  3  FROM t
  4  WHERE d BETWEEN to_date('20140201','YYYYMMDD') AND to_date('20140228','YYYYMMDD');

SQL> EXPLAIN PLAN SET STATEMENT_ID = 'n' FOR
  2  SELECT *
  3  FROM t
```

```
  4  WHERE n BETWEEN 20140201 AND 20140228;

SQL> EXPLAIN PLAN SET STATEMENT_ID = 'c' FOR
  2  SELECT *
  3  FROM t
  4  WHERE c BETWEEN '20140201' AND '20140228';

SQL> SELECT statement_id, cardinality FROM plan_table WHERE id = 0;

STATEMENT_ID CARDINALITY
------------ -----------
d                     29
n                     11
c                     11
```

第五个问题同样与查询优化器有关。但是这一次，是因为隐式转换（作为一个通用规则，始终应该避免隐式转换）。可能发生隐式转换阻止查询优化器选择索引的问题。为了演示这个问题，我使用前面例子中的同一张表。在这张表上，会创建出一个基于VARCHAR2数据类型的列的索引。如果WHERE子句包含对使用字符串的列的限制，查询优化器会选择该索引。然而，如果限制条件上使用了数字（开发人员自己“知道”只有数字值存储在列中……），则会使用全表扫描（注意，在第二个SQL语句中，基于to_number函数的隐式转换阻止了该索引的使用），因此查询优化器就正常地忽略了该索引。

```
SQL> CREATE INDEX i ON t (c);

SQL> SELECT /*+ index(t) */ *
  2  FROM t
  3  WHERE c = '20140228';

---------------------------------------------
| Id  | Operation                   | Name |
---------------------------------------------
|   0 | SELECT STATEMENT            |      |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |
|*  2 |   INDEX RANGE SCAN          | I    |
---------------------------------------------

   2 - access("C"='20140228')

SQL> SELECT /*+ index(t) */ *
  2  FROM t
  3  WHERE c = 20140228;

------------------------------------
| Id  | Operation          | Name |
------------------------------------
|   0 | SELECT STATEMENT   |      |
|*  1 |  TABLE ACCESS FULL| T    |
------------------------------------

   1 - filter(TO_NUMBER("C")=20140228)
```

概括起来，你有充足的理由去选择正确的数据类型。这样做可能会为你省去一大堆问题。

16.2.2　数据类型选择最佳实践

正如上一节中讨论的那样，核心原则是每种数据类型都应该只存储它们被设计用来存储的值。举个例子，数字必须存储在数字数据类型中，而不是字符串数据类型中。此外，一旦存在多种数据类型（例如，几种数据类型都可以存储字符串）可选时，最重要的原则就是所选择的数据类型，能够以最高效的方式完整地存储数据。换句话说，应该避免丢失信息或性能。

接下来的部分会提供一些选择数据类型时应该考虑的信息（主要与性能有关）。这些部分涵盖四个主要的内置数据类型类别：数字、字符串、比特串以及日期时间。

1. 数字

用来存储浮点型数值和整型数值的主要数据类型是NUMBER。这是一种可变长的数据类型。这意味着可以通过精度和小数范围，来指定用于存储数据的精确度。一旦这种数据类型用于存储整数，或一旦完整的精确度没有必要，一定要记得指定范围以节省空间。下面的例子展示相同的输入值，根据不同的小数范围，是如何被舍入为21个字节或2个字节的：

```
SQL> CREATE TABLE t (n1 NUMBER, n2 NUMBER(*,2));

SQL> INSERT INTO t VALUES (1/3, 1/3);

SQL> SELECT * FROM t;

                                       N1  N2
----------------------------------------- ---
.3333333333333333333333333333333333333333 .33

SQL> SELECT vsize(n1), vsize(n2) FROM t;

VSIZE(N1) VSIZE(N2)
--------- ---------
       21         2
```

因为内部格式是专有的，CPU无法使用硬件浮点单元直接处理以NUMBER类型存储的值。取而代之，CPU通过内部的Oracle库程序来处理这些浮点值。因为这个原因，在支撑数值计算负载时NUMBER数据类型不够高效。为解决这个问题，可以使用BINARY_FLOAT和BINARY_DOUBLE。与NUMBER数据类型相比，它们的核心优势是它们实现了IEEE 754标准，因此CPU可以直接处理它们。这两种类型的主要劣势是它们是基于二进制的浮点数。因此，这些类型无法精确地表示经常用于商业和财务应用程序的十进制小数。表16-1总结了这三种数据类型之间的关键不同点。

表16-1　数字数据类型比较

属　　性	NUMBER（精度，小数范围）	BINARY_FLOAT	BINARY_DOUBLE
取值范围	± 1.0E126	± 3.40E38	± 1.79E308
长度	1~22 字节	4 字节	8 字节
支持正负无穷大	是	是	是
支持 NAN	否	是	是

（续）

属　性	NUMBER（精度，小数范围）	BINARY_FLOAT	BINARY_DOUBLE
优势	精确性 可以指定精度和小数范围	速度 固定长度	速度 固定长度

2. 字符串

有三种基本的数据类型用于存储字符串：VARCHAR2、CHAR和CLOB。前两种最高分别支持4000和2000个字节（注意最大长度是按字节指定的，不是字符）。第三种最高支持几TB的数据（实际值取决于默认的块大小）。VARCHAR2与CHAR之间最大的不同是前者为可变长类型，而后者是固定长度的。这意味着CHAR通常用于字符串长度已知的情况下。但是，我的建议是全部使用VARCHAR2，因为它提供比CHAR类型更好的性能。只有在预期字符串的长度要比VARCHAR2支持的最大长度还要大的时候，才使用CLOB类型。从11.1版本开始，CLOB的存储方法有两种：basicfile和securefile。出于性能的考虑，应该使用securefile。

当使用VARCHAR2和CHAR数据类型时，不应该将其最大长度设置为没有必要的长度。这是因为在某些情况下，即使可能没有用到全部空间，数据库引擎也会不得不分配足够的内存来存储你指定的最大长度。因此，可能会有大量的内存被分配却完全用不到。

这三种基本的数据类型按照数据库字符集存储字符串。此外，其他的三种数据类型，NVARCHAR2、NCHAR和NCLOB，可以用于按照国际字符集存储字符串（在数据库级别定义的第二种Unicode字符集）。这三种数据类型和对应的同名基本类型有着相同特征。只有它们的字符集不同。

LONG是另一种字符串数据类型，为支持CLOB已经不推荐使用了。你不应该再使用它；提供这种类型仅是出于向后兼容性的考虑。

警告　从12.1版本开始，可以将max_string_size初始化参数的值设置为extended，以便将VARCHAR2、NVARCHAR2和RAW类型的最大长度增加至32 767字节。这样做的缺点是数据库引擎会静默地采用LOB数据类型来支持这种更大的最大长度。我的建议是保持max_string_size初始化参数的值为默认设置（standard），如果需要更大的空间，请显式使用LOB数据类型。

---

3. 比特串

有两种数据类型用于存储比特串：RAW和BLOB。第一种最高支持2000个字节。只有在预计比特串大于2000个字节时，才应该使用第二种类型。从11.1版本开始，BLOB的存储方法有两种：basicfile和securefile。出于性能的考虑，应该使用securefile。

另一种比特串数据类型是LONG RAW，但是为了支持BLOB已经不推荐使用。你不应该再使用它；提供这种类型仅出于向后兼容性的考虑。

4. 日期时间

用于存储日期时间值的数据类型有DATE、TIMESTAMP、TIMESTAMP WITH TIME ZONE以及TIMESTAMP WITH LOCAL TIME ZONE。这几种类型都会存储的信息如下：年、月、日、小时、分钟以及秒。这部分的长度

固定在7个字节。三种基于TIMESTAMP的数据类型可能还会存储秒的小数部分（0~9位数字，默认6位）。这部分是可变长度：0~4字节。最后，TIMESTAMP WITH TIME ZONE使用两个额外的字节存储时区。因为它们全部都存储不同的信息，存储所需数据占用空间最少的那一种，就是最合适的数据类型。

16.3 行迁移和行链接

迁移和链接的行经常被搞混。依我看来，主要的原因有两个。第一，两者有共同的特性，所以很容易混淆。第二，Oracle在它的文档以及软件实现当中，在如何区分两者这一点上从来没有非常一致过。所以，在描述如何发现和避免行迁移和行链接之前，很有必要简单描述一下两者之间的区别。

16.3.1 迁移与链接

将记录插入到一个块中时，数据库引擎会保留一些空闲空间以供未来更新使用。可以通过使用PCTFREE参数来定义为更新保留的空闲空间总量。为了演示这个参数，我在图16-3描绘的块中插入了六行数据。因为达到了通过PCTFREE设置的阈值，对于这个块来讲不再能够插入数据。

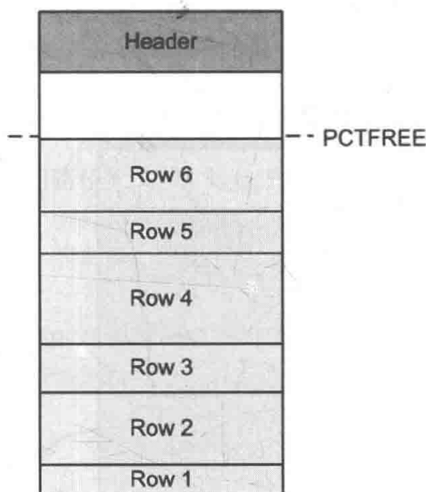


图16-3 插入时会留出一些空闲空间供未来更新

更新一行记录并且其长度增加时，数据库引擎会尝试在存储它的块中寻找足够的空闲空间。当没有足够的空闲空间可用时，会将此行数据分为两个片断。第一个片断（只包含控制信息，比如指向第二个片断的rowid）保留在原始的块中。这对于避免变更rowid来说是有必要的。理解这一点至关重要，因为rowid不仅会被数据库引擎永久地存储在索引中，而且也会被客户端应用程序临时存储在内存中。第二个片断，包含所有的数据，进入到另一个块当中。这种类型的行称为迁移的行。例如，在图16-4中，行4就被迁移了。

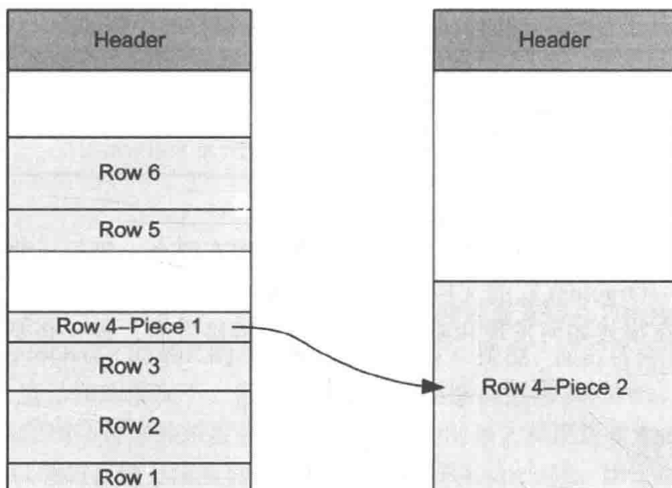


图16-4　更新后的行因为无法继续存储在原始的块中而被迁移至另一个块中

当一行数据太大而无法放入到一个单独的块中，它就会被分为两个或更多的片断。然后，每个片断都被存储在一个不同的块中，此时在各个片断之间就会建立起一个链接。这种类型的行称为链接的行。为了演示，图16-5展示了一个链接了三个块的行。

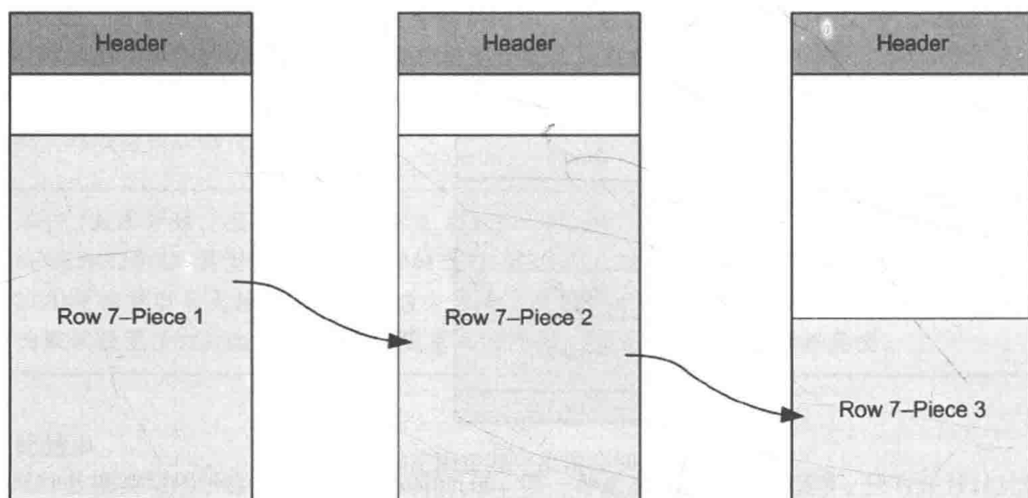


图16-5　一个链接的行被分为两个或更多的部分

还有另外一种情况会引起行链接：拥有超过255个列的表。实际上，数据库引擎无法在一个单独的行片断中存储超过255个列。因此，一旦需要存储的列超过255个时，这个行就会分裂成几个片断。这是一种特殊情况，这几个属于同一行的片断也可以存储在一个单独的块中。这称为块内行链接。图16-6展示了一个拥有三个片断（因为它有654个列）的行。

注意，迁移的行是由更新引起，而链接的行是由插入或者更新引起。当链接的行是由更新引起时，迁移和链接可能会同时发生在这些行上。

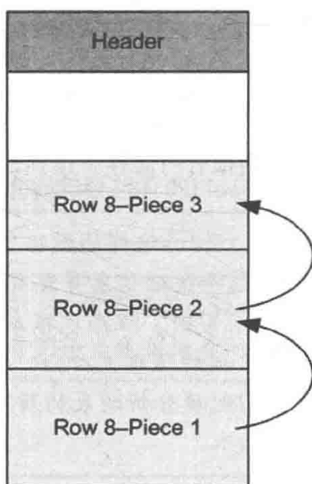


图16-6 拥有超过255列的表可能会引起块内行链接

16.3.2 问题描述

由行迁移引起的性能影响取决于读取行所使用的访问路径。如果迁移的行是通过rowid进行访问的，则成本加倍。事实上，需要分别访问两个行片断。相反，如果它们是通过全表扫描进行访问的，则没有什么开销。这是因为第一个行片断不包含数据，会被直接跳过。

由行链接引起的性能影响与访问路径无关。事实上，每次找到第一个行片断时，都有必要通过rowid访问其他所有的片断。但是，有一个例外情况。正如之前在16.1节中讨论过的，当只需要行的一部分时，可能不需要访问所有的片断。举例来说，如果只需要在第一个片断中存储的列，则没有必要访问其他所有的片断。

与行迁移、行链接两者都有关联的开销与行级锁有关系。必须锁定每一个行片断。这意味着由锁定引起的开销会随片断的数量成比例增加。

16.3.3 问题识别

主要有两种方法用于检测迁移和链接的行。遗憾的是，两者都不是基于响应时间的。这意味着没有关于该问题带来的真正影响的信息。第一种方法，是基于v$sysstat和v$sesstat视图，而且仅仅是给出一个线索提示数据库中的某处存在着迁移或链接的行。其思路是检查名为table fetch continued row的统计信息，该统计信息能够给出读取超过一个以上行片断（包括块内链接的行）的提取操作的数量。这个统计信息也可以与table scan rows gotten以及table fetch by rowid作对比，以便评估行链接和行迁移的相对影响。

比较起来，第二种方法会给出关于迁移的和链接的行的精确信息。遗憾的是，它要求为每张潜在包含链接的或迁移的行的表执行ANALYZE TABLE LIST CHAINED ROWS语句。如果找到链接的或迁移的行，会将它们的rowid插入到chained_rows表中。然后，如下面的查询所示，根据这些rowid可以估算这些行的大小，基于此，再将这些行的大小与块大小进行比较，就可以识别出它们是否是迁移或链接的行。

```
SELECT vsize(<col1>) + vsize(<col2>) + ... + vsize(<coln>)
FROM <table>
WHERE rowid = '<rowid>'
```

另外作为一种选择，也可以查看类似dba_tables这样的视图中的avg_row_len列，当做粗略估算。如果平均行大小接近或甚至大于块大小，则很有可能存在链接的行。

警告　正如在第8章中讨论过的，类似dba_tables这样的数据字典视图中的chain_cnt列本应提供链接的和迁移的行数量。遗憾的是，这个统计信息并没有被dbms_stats包收集。如果没有设置chain_cnt，程序包会将它设置为0。否则，程序包根本不会修改chain_cnt。chain_cnt.sql脚本演示了这种行为。尽管使用正确的值填充chain_cnt的唯一途径是执行ANALYZE TABLE COMPUTE STATISTICS语句，但这会引起被分析的表的所有对象统计信息都被覆盖掉。所以不推荐这种做法。

16.3.4　解决方案

适用于避免迁移的对策和适用于避免链接的那些对策有所不同。因此，我强调一下，在采取措施之前，必须查明问题是由迁移还是链接引起的。

预防行迁移是可行的。这只是正确设置PCTFREE的问题，或者换句话说，在原始的块中保留足够的空闲空间，就能解决完整存储修改的行的问题。这种方式下，如果已经断定正在遭遇行迁移，就应该增加当前PCTFREE的值。你应该估算平均的行增长幅度以便选择一个合理的值。要达到这个目标，你应该知道这些行被插入时的平均大小，以及当它们不再被更新的时候的平均大小。

要从一张表中移除迁移的行，有两种可能性。首先，可以通过导出/导入或ALTER TABLE MOVE来彻底整理这张表。其次，可以只将迁移的行复制到一张临时表中，然后在原始表中将它们删除并重新插入。第二种方法在只有一小部分的行被迁移，而且没有足够的时间或资源来彻底整理一张表的时候尤其有用。

避免行链接则要困难得多。显而易见的解决方案是使用更大的块大小。然而有时候，即使最大的块大小也是不够大的。此外，如果链接是因为列的数量超过了255，那么只有重新设计才有用。因此，在某些情况下，这个问题唯一可行的解决方案，是将不经常访问的列放置在表的末尾，从而避免每次都扫描所有的行片断。

16.4　块争用

块争用，会在多个进程同一时间争相访问相同的块时出现，能够导致应用程序性能低下。块争用有时可以通过操纵表或索引的物理存储参数来减轻。本节会讲述在哪些情况下应用程序会遭遇块争用，并会介绍如何识别和预防这个问题。

16.4.1　问题描述

缓冲区缓存在属于同一数据库实例的所有进程之间共享。因此，多个进程可能会需要同时读取或

修改在缓冲区缓存中存储的相同块。为避免访问冲突，每个进程在能够访问缓冲区缓存中的块之前，必须在这个块上持有一个Pin（这个规则也有例外的情况，但是对于本节的目的而言，讨论它们并不重要）。Pin是一种短暂锁，被进程以共享或独占模式持有。在一个给定的块上，可能会有多个进程以共享模式持有Pin（例如，如果它们都只是想读取这个数据块），而只能有一个单独的进程能够以独占模式持有Pin（需要修改此块）。一旦有进程想要持有的Pin与其他进程持有的Pin冲突，它就必须进入等待。此时这个进程就会面临块争用。

注意 在能够Pin或取消Pin一个块之前，进程必须获得保护该块的缓存缓冲区链的闩锁。由于这个原因，可能会出现块争用被闩锁争用掩盖或伴随着闩锁争用一起发生的情况。

16.4.2 问题识别

如果遵从本书第二部分提供的建议，识别块争用问题唯一有效的方式是衡量因块争用损失了多少时间。出于这个目的，应该检查应用程序是否遭遇了与块争用相关的等待事件，也就是buffer busy waits。实际上，遭遇块争用的进程会等待这个事件。因此，如果这个事件作为相关组件出现在资源使用率配置文件中，那么这个应用程序正在遭受块争用带来的影响。在这种情况下要排查故障，需要以下信息：

- 遭遇等待事件的SQL语句
- 等待出现在哪个段上
- 等待是在哪种类型的块上发生的

正如在第二部分中描述的，获取所需信息的最佳方式取决于问题种类。问题是可重现的还是不可重现的？对于不可重现的问题，分析是实时执行的还是事后执行的？此外，还应该考虑授权需求，例如，你有没有Diagnostic Pack诊断包的授权。还有一点很重要，就是要认识到并非第二部分中描述的所有技巧都适合精确诊断块争用问题。事实上，尽管可以通过使用基于动态性能视图的技术（例如，Snapper或Active Session History）和SQL跟踪明确地识别块争用问题，但当你遇到涉及某些不确定性的情况时，使用基于AWR和Statspack报告的技巧就不行了。这是因为有以下两个主要的原因。

- 系统级别的分析只能精确到与影响整个系统有关的问题。因此，你可能会错过只影响几个会话的问题。
- AWR和Statspack报告基于一组动态性能视图所提供的信息，而这些动态性能视图无法一直保持收集数据。为了验证这个限制，我们看一下v$waitstat视图（下面的查询作为一个例子显示了它提供的信息）。尽管这个视图的内容提供了所需的关于等待出现在哪些类型的块上的信息，但是还是没有办法确定有哪些SQL语句在等待这些块（注意这个视图中的所有列都被显示出来了）。

```
SQL> SELECT * FROM v$waitstat;

CLASS                   COUNT       TIME
------------------ ---------- ----------
data block             102011       5162
```

```
sort block                 0        0
save undo block            0        0
segment header         76053      719
save undo header           0        0
free list               3265       12
extent map                 0        0
1st level bmb           6318      352
2nd level bmb            185        3
3rd level bmb              0        0
bitmap block               0        0
bitmap index block         0        0
file header block        389     2069
unused                     0        0
system undo header         1        1
system undo block          0        0
undo header             3244       70
undo block                38        2
```

注意　关于v$waitstat视图需要理解的核心内容是关于块类型的class列，而不是发生等待的数据类型或结构。例如，如果争用是因为包含在段头块中的自由列表引起的，则报告会体现等待是发生在segment header类下，而不是在free list类下。事实上，free list适用于只存储自由列表信息的数据块（这样的块是在将FREELIST GROUPS的值设置为大于1时被创建出来的）。另一个例子是关于索引的。如果存储索引的块发生争用，则会在data block类下报告等待。

根据刚刚解释的原因，接下来的两部分会提供使用SQL跟踪和v$session视图（基于Snapper；在本例中，Active Session History不是很合适，因为我使用的测试只会持续运行几秒钟）识别问题的例子。在这些列子中使用的块争用是通过buffer_busy_waits.sql脚本生成的。

1. 使用 SQL跟踪

我建议使用TVD$XTAT来处理buffer_busy_waits.sql脚本生成的跟踪文件。尽管可以选择使用TKPROF或TVD$XTAT中的任何一个，但我还是推荐后者。这是因为TKPROF不会为你提供排查块争用问题所需的全部信息。尤其是，它不提供关于遭遇buffer busy waits的块的信息。

由TVD$XTAT为当前的例子生成的输出文件，以及它依赖的跟踪文件，都在buffer_busy_waits.zip文件中提供，这些文件显示其中一条SQL语句（即一条UPDATE语句）几乎是整个响应时间的元凶。下面的摘录显示该UPDATE语句的执行统计信息。通过10 000次执行，测量出消耗的时间为6.187秒，其中CPU时间为2.411秒。

```
UPDATE /*+ index(t) */ T SET D = SYSDATE WHERE ID = :B1 AND N10 = ID

Call     Count Misses   CPU Elapsed PIO    LIO Consistent Current   Rows
------- ------ ------ ----- ------- --- ------ ---------- ------- ------
Parse        1      1 0.001   0.000   0      0          0       0      0
Execute 10,000      1 2.410   6.186   0 73,084     43,797  29,287 10,000
Fetch        0      0 0.000   0.000   0      0          0       0      0
------- ------ ------ ----- ------- --- ------ ---------- ------- ------
Total   10,001      2 2.411   6.187   0 73,084     43,797  29,287 10,000
```

因为CPU时间只占响应时间的39%，有必要看一下处理过程中出现的等待，以便找出时间是如何花费掉的。下面的摘录精确地显示了这个信息。你可以看到最大的消耗者，是花费了3.322秒的buffer busy waits。还要注意，在这个特定的例子中，cache buffers chains闩锁拥有最小的争用。

```
                             Total            Number of Duration per
Component                 Duration       %    Events        Event
-------------------------- -------- ------- --------- ------------
buffer busy waits            3.322  53.843    10,953        0.000
CPU                          2.410  39.056       n/a          n/a
latch: In memory undo latch  0.278   4.509     6,389        0.000
latch: cache buffers chains  0.158   2.559     6,238        0.000
recursive statements         0.001   0.016       n/a          n/a
enq: HW - contention         0.000   0.008         1        0.000
Disk file operations I/O     0.000   0.007         1        0.000
latch free                   0.000   0.001         1        0.000
-------------------------- -------- -------
Total                        6.170 100.000
```

在TVD$XTAT中提供的额外信息，但在TKPROF的输出中却缺失的信息，是一个包含着在哪些块上出现了等待的列表。下面的摘录显示，在此处所分析的案例中，这样一个列表看起来是什么样子的。你可以看到超过99%的buffer busy waits出现在文件4中的编号为836 775的块上。还要注意遭遇争用的块只是一个数据块。

```
File   Block   Total            Number of         Duration per
Number Number  Duration     %      Events      %        Event Class
------ ------- -------- ------- --------- ------- ------------ ---------------
4      836,775    3.290  99.045     9,779  89.281          336 data block (1)
3      272        0.006   0.173       196   1.789           29 undo header (35)
3      192        0.003   0.094       108   0.986           29 undo header (25)
3      128        0.003   0.094       117   1.068           27 undo header (17)
3      256        0.003   0.090       122   1.114           24 undo header (33)
3      144        0.003   0.088        88   0.803           33 undo header (19)
3      208        0.003   0.083        99   0.904           28 undo header (27)
3      240        0.003   0.083       112   1.023           24 undo header (31)
3      176        0.003   0.082       105   0.959           26 undo header (23)
3      224        0.003   0.081       107   0.977           25 undo header (29)
...
------ ------- -------- ------- --------- ------- ------------
Total             3.322 100.000    10,953 100.000          303
```

根据此信息，可以通过以下查询找出发生等待的段的名称（小心，这个查询的执行可能会占用大量资源）：

```
SQL> SELECT owner, segment_name, segment_type
  2  FROM dba_extents
  3  WHERE file_id = 4
  4  AND 836775 BETWEEN block_id AND block_id+blocks-1;

OWNER SEGMENT_NAME SEGMENT_TYPE
----- ------------ ------------
CHRIS T            TABLE
```

概括起来，整个分析提供以下信息。

- 遭遇等待的SQL语句是一个UPDATE语句。
- 大多数时候，等待出现在一个单独的数据块上。
- 发生等待的段是UPDATE 语句中被更新的表。

2. 使用 Snapper

通过Snapper，可以根据几个准则收集数据。但是，假如你已经识别出必须要排查故障的会话，最好通过执行类似下面的例子中展示的这条命令着手。在本例中，我指定针对所有会话执行buffer_busy_waits.sql脚本。就像输出显示的那样，一条SQL语句（9bjs886z43g7k）不仅消耗了大部分的数据库时间（在采样期间，总计有7个活跃的会话），而且在此情况下，它还遭遇了大量的buffer busy waits。

```
SQL> @snapper.sql ash=sql_id+wait_class+event 1 1 user=chris

-----------------------------------------------------------------
Active% | SQL_ID          | WAIT_CLASS   | EVENT
-----------------------------------------------------------------
   560% | 9bjs886z43g7k | Concurrency | buffer busy waits
   100% | 9bjs886z43g7k | ON CPU      | ON CPU
    40% | 091f2847g34rm | ON CPU      | ON CPU
    20% | 48gc5511n38a1 | ON CPU      | ON CPU
    20% | 9bjs886z43g7k | Concurrency | latch: cache buffers chains
    20% | 9bjs886z43g7k | Concurrency | latch: In memory undo latch
    10% | 93053g60rwz0x | ON CPU      | ON CPU
    10% |               | ON CPU      | ON CPU
    10% | cktrdz5u39r04 | ON CPU      | ON CPU
    10% | 93053g60rwz0x | Concurrency | latch: In memory undo latch
```

接下来，可以使用dbms_xplan包来获取问题SQL语句的文本和执行计划。不出所料，正是上一小节中识别出的同一条UPDATE语句。

```
SQL> SELECT * FROM table(dbms_xplan.display_cursor('9bjs886z43g7k', 0, 'basic'));

UPDATE /*+ index(t) */ T SET D = SYSDATE WHERE ID = :B1 AND N10 = ID
-----------------------------------------------
| Id  | Operation                     | Name |
-----------------------------------------------
|   0 | UPDATE STATEMENT              |      |
|   1 |  UPDATE                       | T    |
|   2 |   TABLE ACCESS BY INDEX ROWID| T    |
|   3 |    INDEX UNIQUE SCAN          | T_PK |
-----------------------------------------------
```

为了深入调查buffer busy waits，可以再次执行Snapper，就像下面例子中这样，但是这一次使用一组参数执行它，以便显示关于buffer busy waits的详细信息。注意，对于buffer busy waits，与这个事件有关的参数有以下含义：p1是文件号（4），p2是块号（836 775），还有p3是遭遇等待的块类（1=数据块）。

```
SQL> @snapper.sql ash=event+p1+p2+p3 1 1 user=chris

---------------------------------------------------------------------------
Active% | EVENT                           | P1          | P2          | P3
---------------------------------------------------------------------------
```

```
560% | buffer busy waits             | 4          | 836775     | 1
190% | ON CPU                        |            |            |
 20% | latch: cache buffers chains   | 1992028296 | 155        | 0
 20% | latch: In memory undo latch   | 1966009168 | 251        | 0
 10% | latch: In memory undo latch   | 1966009648 | 251        | 0
```

概括起来，使用Snapper的分析准确地描述了在使用SQL跟踪时识别出的一模一样的问题。

16.4.3 解决方案

通过识别遭遇等待的SQL语句、块类以及段，应该能够识别出问题的根本原因。接下来我们来讨论一些关于常见块类的典型案例。

1. 数据块的争用

所有用来组成表或索引段且不用于存储元数据（例如段头）的块称作数据块。它们的争用源自两个主要原因。第一个是在给定的段上高频率的数据块访问。第二个是高频率的执行。乍一看，这两个是同一回事。为什么它们实际上是不同的，需要做出一些解释。在第一种情况中，问题的起源是低效率的执行计划引起对某些块执行高频率的数据块访问。通常，这是因为低效率的关联组合操作（例如，嵌套循环链接）引起。在此情况下，即使只有两条或三条SQL语句并发执行也有可能引发争用。相反，在第二种情况中，问题的起源是在同一时刻非常频繁地执行访问同一个块的多条SQL语句。换句话说，针对（少量）块执行SQL语句的并发数量是问题所在。也有可能出现两者同时发生的情况。如果两个问题同时出现，在处理第二个问题之前要处理好第一个问题。实际上，当第一个问题消失的时候，第二个问题可能也就不见了。

要解决第一个问题，有必要进行SQL优化。必须产生一个高效的执行计划以取代低效的那个。当然了，在某些情况下，说起来容易做起来难。不管怎样，这的确是你必须要解决的事情。

要解决第二个问题，有多种途径。要使用哪种途径取决于SQL语句的类型（即DELETE、INSERT、SELECT[①]以及UPDATE）和段的类型（即表或索引）。但是，在开始之前，当执行的频率很高的时候，你应该总是问这样一个问题：是不是真的有必要如此频繁地针对相同的数据执行那些SQL语句？实际上，应用程序（例如，实现了某种轮询的应用）过于频繁地执行没有必要的相同SQL语句并不是稀奇的事。如果无法降低执行的频率，则存在以下可能性。注意，在大多数情况下，方法是通过大量的块来分散这些活动以解决此问题。唯一的例外是当多个会话等待相同行的时候。

- 如果在一张表的块上出现争用是因为DELETE、SELECT以及UPDATE语句，应该减少每个块上的行数量。注意这样做与通常在每个块中填入尽可能多数量的行的最佳实践相反。要在每个块中存储更少的行，要么使用较高的PCTFREE，要不然就使用较小的块大小。
- 如果在一张表的块上出现争用是因为INSERT语句，并且使用了自由列表段空间管理，则可以增加自由列表的数量。事实上，使用多个自由列表的目的，恰恰是通过多个块来分散并发执行的INSERT语句。另外一个可行的方法是将段迁移到使用自动段空间管理的表空间中。
- 如果在索引块上出现了争用，有两种可行的解决方案。第一个，可以使用REVERSE选项创建索引。但是注意，如果争用出现在索引的根块上，这个方法一点用也没有。第二个，索引可以

① SELECT语句在两种情况下修改块：第一，当指定了FOR UPDATE选项时；第二，当延迟块清除出现时。

使用散列分区（或子分区），散列要基于索引键的前导列（这样会创建多个根块，所以能够缓解访问一个单独的分区时根块争用的情况）。

关于反转索引值得注意的是，在这些索引上执行的范围扫描无法应用基于范围条件的限制条件（例如，BETWEEN、>或<=）。当然，等价谓词是受支持的。下面的例子，来自reverse_index.sql脚本，展示了在使用REVERSE选项重建索引后，查询优化器不再继续使用该索引的情形：

```
SQL> SELECT * FROM t WHERE n < 10;

-------------------------------------------
| Id  | Operation                   | Name |
-------------------------------------------
|   0 | SELECT STATEMENT            |      |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |
|*  2 |   INDEX RANGE SCAN          | T_I  |
-------------------------------------------

   2 - access("N"<10)

SQL> ALTER INDEX t_i REBUILD REVERSE;

SQL> SELECT * FROM t WHERE n < 10;
-----------------------------------
| Id  | Operation          | Name |
-----------------------------------
|   0 | SELECT STATEMENT   |      |
|*  1 |  TABLE ACCESS FULL| T    |
-----------------------------------

   1 - filter("N"<10)
```

注意，在这种情况下hint也不会有帮助。数据库引擎就是无法在一个反转索引上通过索引范围扫描应用范围条件。因此，下面的例子证实了如果你尝试强制查询优化器使用索引访问，则优化器会使用索引全扫描：

```
SQL> SELECT /*+ index(t) */ * FROM t WHERE n < 10;

-------------------------------------------
| Id  | Operation                   | Name |
-------------------------------------------
|   0 | SELECT STATEMENT            |      |
|   1 |  TABLE ACCESS BY INDEX ROWID| T    |
|*  2 |   INDEX FULL SCAN           | T_I  |
-------------------------------------------

   2 - filter("N"<10)
```

2. 段头块的争用

每个表和索引段都有一个段头块。这个块包含以下元数据：关于段的高水位线的信息、组成段的扩展的列表，以及关于空闲空间的信息。为管理空闲空间，段头块包含（取决于使用的段空间管理类型）自由列表或一个包含着自动段空间管理信息的块的列表。通常，当多个进程同时修改段头块的内

容时会遭遇争用。注意，在以下情况下会修改段头块：

- 如果INSERT语句使得段头块有必要增加高水位线；
- 如果INSERT语句使得段头块有必要分配新的扩展；
- 如果DELETE、INSERT和UPDATE语句使得段头块有必要修改自由列表。

针对以上这些情况，一个可能的解决方案是给这个段分区，以便将负载分散至多个段头块上。大多数时候，这可以通过散列分区来实现，尽管这样，根据负载和分区键的不同，其他的分区方法也可能是可行的。但是，如果问题是由第二或第三种情况引起的，则存在其他解决方案。对于第二种，应该直接使用更大的扩展。通过这种方式，很少会分配新的扩展。对于第三种，借助于自由列表组，可以将自由列表移动到其他块中，但这不适用于使用自动段空间管理的表空间。事实上，当使用了多个自由列表组时，自由列表就不再位于段头块中（它们被分散到由参数FREELIST GROUPS指定的值相等数量的块中，这样就可以期待有较少的争用出现在它们头上，你不能简单地将争用转移至其他地方）。另一种可能性是使用自动段空间管理的表空间代替自由列表段空间管理。

注意 自由列表组只有在使用真实应用程序群集时才有用，这是Oracle数据库领域一个广为流传的神话。这是**错误的**。自由列表组在每种数据库中都有用。我强调这一点是因为曾经读到或听到过太多关于这个问题的错误描述了。

---

3. Undo头和Undo块的争用

这些类型的块争用会在两种情况下出现。第一种，仅对于undo头块来说，是当只有少量undo段可用而且并发提交（或初始化或回滚）大量事务的时候。这个问题只有在使用手动undo管理的时候才会出现。换句话说，这种情况一般只有在数据库管理员手动创建回滚段时才会发生。要解决这个问题，应该使用自动undo管理。第二种是当多个会话在同一时间修改并查询相同块的时候。此时会导致数据库引擎创建大量的一致性读块，并且因此需要同时访问这个块以及与它关联的undo块。在这种情况下可以做的事情很少，只有减少数据块的并发性，继而同时减少undo块的并发性。

4. 扩展映射块的争用

正如在之前的“段头块的争用”部分中讨论的，段头块包含一个组成段的扩展的列表。如果该列表在段头中装不下，它就会分散在多个块中：段头块加上一个或多个扩展映射块。*扩展映射块*（extent map block）会在并发执行INSERT语句导致频繁分配新的扩展的时候遭遇争用。要解决这个问题，应该使用更大的扩展。

5. 自由列表块的争用

正如在之前的“段头块的争用”部分中讨论的，借助于自由列表组，可以将自由列表移动到其他块中，这些块称作*自由列表块*。当并发DELETE、INSERT或UPDATE语句导致频繁修改自由列表时，自由列表块就会遭遇争用。要解决这个问题，应该增加自由列表组的数量。另外一种可能性是使用自动段空间管理的表空间代替自由列表段空间管理。

16.5 数据压缩

通常压缩数据的目标是节省磁盘空间。既然我们正在考虑性能问题，本节会介绍数据压缩经常被遗忘的另一个优势：缩短响应时间。

16.5.1 概念

利用数据压缩来实现更好性能的思想源于一个简单的概念。如果一个SQL语句必须通过全表（或分区）扫描处理大量的数据，则资源使用的大户很有可能就是与磁盘I/O相关的操作。在这种情况下，降低从磁盘读取的数据总量将会改进性能。实际上，性能应该按压缩因子成比例增加。下面来自data_compression.sql脚本的例子，证实了这一点：

```
SQL> CREATE TABLE t NOCOMPRESS AS
  2  WITH
  3    t AS (SELECT /*+ materialize */ rownum AS n
  4            FROM dual
  5            CONNECT BY level <= 1000)
  6  SELECT rownum AS n, rpad(' ',500,mod(rownum,15)) AS pad
  7  FROM t, t, t
  8  WHERE rownum <= 1E7;

SQL> execute dbms_stats.gather_table_stats(ownname=>user, tabname=>'t')

SQL> SELECT table_name, blocks FROM user_tables WHERE table_name = 'T';

TABLE_NAME BLOCKS
---------- ------
T          715474

SQL> SELECT count(n) FROM t;

  COUNT(N)
----------
  10000000

Elapsed: 00:00:27.91

SQL> ALTER TABLE t MOVE COMPRESS;

SQL> execute dbms_stats.gather_table_stats(ownname=>user, tabname=>'t')

SQL> SELECT table_name, blocks FROM user_tables WHERE table_name = 'T';

TABLE_NAME BLOCKS
---------- ------
T          140367

SQL> SELECT count(n) FROM t;

  COUNT(N)
```

```
----------
  10000000

Elapsed: 00:00:05.38

SQL> SELECT 715474/140367, 27.91/05.38 FROM dual;

715474/140367 27.91/05.38
------------- -----------
   5.09716671  5.18773234
```

为了在完全扫描操作中利用数据压缩，正如在之前的例子中所示（换句话说，为了使完全扫描操作运行得更快），可能有必要节约CPU资源。这不是因为"解压缩"这些块时的CPU负载（其实此负载很小，因为默认的压缩基于一种非常简单的算法，只会对重复的列值去重），而是因为由SQL引擎执行的操作（在之前的例子中，是访问这些块和计数的操作）会在更短的时间内执行。还要注意，减少物理I/O操作的数量也可以减少CPU消耗。例如，在我的测试系统上，在执行测试查询期间，不使用压缩时会话级别的平均CPU使用率大约是18%，而使用压缩后大约是27%。

16.5.2 要求

Oracle数据库企业版（而非其他任何版本）提供多种数据压缩方法。此外，其中一部分算法只在特定的版本中可用；其他的版本受授权的问题限制。表16-2总结了各个版本都提供了哪些方法，以及使用它们的授权要求。

表16-2 由Oracle数据库提供的压缩方法

方 法	版 本	授权要求
基础表压缩	从 9.2 开始	无
高级行压缩（也就是OLTP表压缩）	从 11.1 开始	高级压缩选项
混合列压缩	从 11.2 开始	包含数据的表空间必须迁移至Exadata存储、ZFS存储或Pillar Axiom 600存储中

一种数据压缩方法是否能应用还取决于准备进行压缩的表的实现。具体来说，有以下几个限制。

- 只有堆表可以被压缩（索引组织表、外部表以及属于群集的一部分表都不受支持）。
- 除了混合列压缩，压缩的表不能拥有超过255个列。
- 压缩的表不能拥有LONG或LONG RAW类型的列。
- 压缩的表不能启用行级别依赖跟踪。
- 压缩的表不能属于sys用户或被存储在system表空间中。

16.5.3 方法

为了对表16-2中提到的三种方法之间的区别进行总体描述，我们来快速看一下它们的关键特性，以及应该在什么时候去应用它们。

基础表压缩是Oracle引入的第一种压缩方法。要使用这种方法，必须通过直接路径接口执行加载。

换句话说，仅在使用以下操作之一时，基础表压缩才会压缩数据块：

- ❑ CREATE TABLE ... COMPRESS ... AS SELECT ...
- ❑ ALTER TABLE ... MOVE COMPRESS
- ❑ INSERT /*+ append */ INTO ... SELECT ...
- ❑ INSERT /*+ parallel(...) */ INTO ... SELECT ...
- ❑ 由应用程序使用OCI直接路径接口执行的加载（例如SQL*Loader实用工具）

为了确保在每个块中尽可能多地存储数据，当使用基础表压缩时，数据库引擎默认将PCTFREE设置为0。假如数据是通过正常的INSERT语句插入的，它会存储在没有压缩的块中。基础表压缩还有一个劣势，就是通常不仅UPDATE语句会使得更新的行存储在没有压缩的块中从而导致行迁移，而且由DELETE语句造成的压缩块中的空闲空间一般也不会被重新利用。出于这些原因，我建议只在（主要用于）只读的段上使用基础表压缩。举例来说，在一个存储着很长历史数据的分区表中，而且只有最近的少数几个分区会被修改，压缩（主要用于）只读分区可能有所帮助。数据集市和完全刷新的物化视图使用基础表压缩也是不错的选择。

引入高级行压缩的主要目的是通过提供接近于基础表压缩（内部存储基本一样）的压缩比例，来支持同时受到普通插入和修改（例如来自UPDATE和DELETE语句的）问题困扰的表。因为这种压缩方法工作的方式是动态的（并不会针对每个INSERT语句或修改都进行数据压缩；相反，会在给定的块中包含足够的未压缩数据时进行数据压缩），很难给出关于它的使用建议。还有几种情况高级行压缩的表现也不优于基础表压缩。此外，使用高级行压缩，会比未压缩的表产生更多的undo和redo。因此，为了弄清楚高级行压缩是否能够正确地处理（多数时候）非只读数据，我强烈建议你首先根据预期的负载仔细地进行测试。

混合列压缩则基于完全不同的技术。关键区别是，对于一个具体行来说不再顺序地存储列，就像图16-1展示的那样。相反，数据是按照一列一列来存储的，结果就是，来自同一行的列可能会存储在不同的块中。此外，为尽可能多地将相同类型的数据存储在一起，基础的存储结构，称之为逻辑压缩单元（logical compression unit），由多个块组成。一列一列的存储数据以及使用更大的存储结构都是为了实现更高的压缩比。然而，当处理压缩的数据时，更高的压缩比通常会关联更高的CPU消耗。出于这个原因，可以在四个压缩级别之间选择（这里是根据预期的压缩比和CPU消耗排序的）：QUERY LOW、QUERY HIGH、ARCHIVE LOW以及 ARCHIVE HIGH。注意在Exadata系统上，可以减轻解压缩的负载（但是需要智能扫描），而压缩通常是由数据库实例执行的。混合列压缩的其他弊端如下所示。

- ❑ 只有当数据是通过直接路径接口加载的时候才会被压缩（与基础表压缩的要求一样）；正常的INSERT语句将数据存储在使用高级行压缩的块中。
- ❑ 在12.1.0.1版本以前（包括12.1.0.1版本），不支持行级别锁定；自12.1.0.2版本起，可控制行级别锁定的使用（默认是不使用）。当不使用行级别锁定时，仅可以锁定整个逻辑压缩单元。
- ❑ 即使在表级别没有显式启用行移动，UPDATE语句还是会导致行移动，因此，rowid会发生变化。

概括起来，我建议只在与基础表压缩相同的情况下使用混合列压缩。唯一的额外要求是你需要一个表16-2中列出的存储子系统。

需要注意的是，正如表16-3所示，在11.1和12.1之间的每个版本都更改了用于激活某一具体数据压缩方法的关键字。同时还有，每个新版本都不赞成之前版本使用的关键字。在所有版本中都以相同方式发挥作用的关键字只有以下两个：

- NOCOMPRESS禁用表压缩
- COMPRESS启用基础表压缩

表16-3 不同的版本支持的不同表压缩子句

版　本	基础表压缩	高级行压缩	混合列压缩
11.1	COMPRESS FOR DIRECT_LOAD OPERATIONS	COMPRESS FOR ALL OPERATIONS	/
11.2	COMPRESS BASIC	COMPRESS FOR OLTP	COMPRESS FOR [QUERY|ARCHIVE] [LOW|HIGH]
12.1	ROW STORE COMPRESS BASIC	ROW STORE COMPRESS ADVANCED	COLUMN STORE COMPRESS FOR [QUERY|ARCHIVE] [LOW|HIGH]

参考文献

Adams, Steve, "Oracle Internals and Advanced Performance Tuning." Miracle Master Class, 2003.

Ahmed, Rafi et al, "Cost-Based Transformation in Oracle." VLDB Endowment, 2006.

Ahmed, Rafi, "Query processing in Oracle DBMS." ACM, 2010.

Lee, Allison and Mohamed Zait, "Closing the query processing loop in Oracle 11g." VLDB Endowment, 2008.

Alomari, Ahmed, *Oracle8i & Unix Performance Tuning*. Prentice Hall PTR, 2001.

Andersen, Lance, *JDBC 4.1 Specification*. Oracle Corporation, 2011.

Antognini, Christian, "Tracing Bind Variables and Waits." SOUG Newsletter, 2000.

Antognini, Christian, "When should an index be used?" SOUG Newsletter, 2001.

Antognini, Christian, Dominique Duay, Arturo Guadagnin, and Peter Welker, "Oracle Optimization Solutions." Trivadis TechnoCircle, 2004.

Antognini, Christian, "CBO: A Configuration Roadmap." Hotsos Symposium, 2005.

Antognini, Christian, "SQL Profiles." Trivadis CBO Days, 2006.

Antognini, Christian, "Oracle Data Storage Internals." Trivadis Traning, 2007.

Bellamkonda, Srikanth et al, "Enhanced subquery optimizations in Oracle." VLDB Endowment, 2009.

Booch, Grady, *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 1994.

Brady, James, "A Theory of Productivity in the Creative Process." IEEE Computer Graphics and Applications, 1986.

Breitling, Wolfgang, "A Look Under the Hood of CBO: the 10053 Event." Hotsos Symposium, 2003.

Breitling, Wolfgang, "Histograms—Myths and Facts." Trivadis CBO Days, 2006.

Breitling, Wolfgang, "Joins, Skew and Histograms." Hotsos Symposium, 2007.

Brown, Thomas, "Scaling Applications through Proper Cursor Management." Hotsos Symposium, 2004.

Burns, Doug, "Statistics on Partitioned Objects." Hotsos Symposium, 2011.

Caffrey, Melanie et al, *Expert Oracle Practices*. Apress, 2010.

Chakkappen, Sunil et al, "Efficient and Scalable Statistics Gathering for Large Databases in Oracle 11g." ACM, 2008.

Chaudhuri, Surajit, "An Overview of Query Optimization in Relational Systems." ACM Symposium on Principles of Database Systems, 1998.

Dageville, Benoît and Mohamed Zait, "SQL Memory Management in Oracle9*i*." VLDB Endowment, 2002.

Dageville, Benoît et al, "Automatic SQL Tuning in Oracle 10*g*." VLDB Endowment, 2004.

Database Language – SQL. ANSI, 1992.

Database Language – SQL – Part 2: Foundation. ISO/IEC, 2003.

Date, Chris, *Database In Depth*. O'Reilly, 2005.

Dell'Era, Alberto, "Join Over Histograms." 2007.

Dell'Era, Alberto, Alberto Dell'Era's Blog (http://www.adellera.it).

Dyke, Julian, "Library Cache Internals." 2006.

Engsig, Bjørn, "Efficient use of bind variables, cursor_sharing and related cursor parameters." Miracle White Paper, 2002.

Flatz, Lothar, "How to Avoid a Salted Banana." DOAG Conference, 2013.

Foote, Richard, Richard Foote's Oracle Blog (http://richardfoote.wordpress.com).

Foote, Richard, "Indexing New Features: Oracle 11g Release 1 and Release 2", 2010.

Geist, Randolf, "Dynamic Sampling." All Things Oracle, 2012.

Geist, Randolf, "Everything You Wanted To Know About FIRST_ROWS_n But Were Afraid To Ask." UKOUG Conference, 2009.

Geist, Randolf, Oracle Related Stuff Blog (http://oracle-randolf.blogspot.com).

Grebe, Thorsten, "Glücksspiel Systemstatistiken – das Märchen von typischen Workload." DOAG Conference, 2012.

Green, Connie and John Beresniewicz, "Understanding Shared Pool Memory Structures." UKOUG Conference, 2006.

Goldratt, Eliyahu, *Theory of Constraints*. North River Press, 1990.

Gongloor, Prabhaker, Sameer Patkar, "Hash Joins, Implementation and Tuning." Oracle Technical Report, 1997.

Gülcü Ceki, *The complete log4j manual*. QOS.ch, 2003.

Hall, Tim, ORACLE-BASE (http://www.oracle-base.com).

Held, Andrea et al, *Der Oracle DBA*. Hanser, 2011.

Hoogland, Frits, "About Multiblock Reads." Hotsos Symposium, 2013.

Jain, Raj, *The Art of Computer Systems Performance Analysis*. Wiley, 1991.

Kolk, Anjo, "The Life of an Oracle Cursor and its Impact on the Shared Pool." AUSOUG Conference, 2006.

Knuth, Donald, "Structured Programming with go to Statements." Computing Surveys, 1974.

Knuth, Donald, *The Art of Computer Programming, Volume 3 – Sorting and Searching*. Addison-Wesley, 1998.

Kyte, Thomas, *Effective Oracle by Design*. McGraw-Hill/Osborne, 2003.

Lahdenmäki, Tapio and Michael Leach, *Relational Database Index Design and the Optimizers*. Wiley, 2005.

Lee, Allison and Mohamed Zait, "Closing The Query Processing Loop in Oracle 11g." VLDB Endowment, 2008.

Lewis, Jonathan, "Compression in Oracle." All Things Oracle, 2013.

Lewis, Jonathan, *Cost-Based Oracle Fundamentals*. Apress, 2006.

Lewis, Jonathan, "Hints and how to use them." Trivadis CBO Days, 2006.

Lewis, Jonathan, Oracle Scratchpad Blog (http://jonathanlewis.wordpress.com).

Lilja, David, *Measuring Computer Performance*. Cambridge University Press, 2000.

Machiavelli Niccoló, *Il Principe*. Einaudi, 1995.

Mahapatra, Tushar and Sanjay Mishra, *Oracle Parallel Processing*. O'Reilly, 2000.

Menon, R.M., *Expert Oracle JDBC Programming*. Apress, 2005.

Mensah, Kuassi, *Oracle Database Programming using Java and Web Services*. Digital Press, 2006.

Merriam-Webster online dictionary (http://www.merriam-webster.com).

Millsap, Cary, "Why You Should Focus on LIOs Instead of PIOs." 2002.

Millsap, Cary with Jeff Holt, *Optimizing Oracle Performance*. O'Reilly, 2003.

Millsap, Cary, *The Method R Guide to Mastering Oracle Trace Data*. CreateSpace, 2013.

Moerkotte, Guido, *Building Query Compilers*. 2009.

Morton, Karen et al, *Pro Oracle SQL*. Apress, 2010.

Nørgaard, Mogens et al, *Oracle Insights*: *Tales of the Oak Table*. Apress, 2004.

Oracle Corporation, "Bug 10050057 - SQL profile not used in the Active Physical Standby (ADG))." Oracle Support note 10050057.8, 2013.

Oracle Corporation, "Bug 13262857 Enh: provide some control over DBMS_STATS index clustering factor computation." Oracle Support note 13262857.8, 2013.

Oracle Corporation, "Bug 14320218 Wrong results with query results cache using PL/SQL function." Oracle Support note 14320218.8, 2013.

Oracle Corporation, "Bug 8328200 - Misleading or excessive STAT# lines for SQL_TRACE / 10046." Oracle Support note 8328200.8, 2012.

Oracle Corporation, "CASE STUDY: Analyzing 10053 Trace Files." Oracle Support note 338137.1, 2012.

Oracle Corporation, "Delete or Update running slow—db file scattered read waits on index range scan." Oracle Support note 296727.1, 2005.

Oracle Corporation, "Deprecating the cursor_sharing = 'SIMILAR' setting." Oracle Support note 1169017.1, 2013.

Oracle Corporation, "EVENT: 10046 'enable SQL statement tracing (including binds/waits).'" Oracle Support note 21154.1, 2012.

Oracle Corporation, "Extra NESTED LOOPS Step In Explain Plan on 11g and Above." Oracle Support note 978496.1, 2013.

Oracle Corporation, "Global statistics - An Explanation." Oracle Support note 236935.1, 2012.

Oracle Corporation, "Handling and resolving unshared cursors/large version_counts." Oracle Support note 296377.1, 2007.

Oracle Corporation, "How to Edit a Stored Outline to Use the Plan from Another Stored Outline." Oracle Support note 730062.1, 2012.

Oracle Corporation, "How To Collect Statistics On Partitioned Table in 10g and 11g." Oracle Support note 1417133.1, 2013.

Oracle Corporation, "How to Monitor SQL Statements with Large Plans Using Real-Time SQL Monitoring?" Oracle Support note 1613163.1, 2014.

Oracle Corporation, "Init.ora Parameter CURSOR_SHARING Reference Note." Oracle Support note 94036.1, 2014.

Oracle Corporation, "Init.ora Parameter OPTIMIZER_SECURE_VIEW_MERGING Reference Note." Oracle Support note 567135.1, 2013.

Oracle Corporation, "Init.ora Parameter PARALLEL_DEGREE_POLICY Reference Note." Oracle Support note 1216277.1, 2013.

Oracle Corporation, "Init.ora Parameter SORT_AREA_RETAINED_SIZE Reference Note." Oracle Support note 30815.1, 2012.

Oracle Corporation, "Init.ora Parameter STAR_TRANSFORMATION_ENABLED Reference Note." Oracle Support note 47358.1, 2013.

Oracle Corporation, "Installing and Using Standby Statspack in 11g." Oracle Support note 454848.1, 2014.

Oracle Corporation, "Interpreting Raw SQL_TRACE output." Oracle Support note 39817.1, 2012.

Oracle Corporation, Java Platform Standard Edition 7 Documentation.

Oracle Corporation, "Master Note for Materialized View (MVIEW)." Oracle Support note 1353040.1, 2013.

Oracle Corporation, "Master Note for OLTP Compression." Oracle Support note 1223705.1, 2012.

Oracle Corporation, "Multi Join Key Pre-fetching." Oracle Support note 264532.1, 2010.

Oracle Corporation, "Real-Time SQL Monitoring." Oracle White Paper, 2009.

Oracle Corporation, "Rolling Cursor Invalidations with DBMS_STATS.AUTO_INVALIDATE." Oracle Support note 557661.1, 2012.

Oracle Corporation, "Rule Based Optimizer is to be Desupported in Oracle10g." Oracle Support note 189702.1, 2012.

Oracle Corporation, "Script to produce HTML report with top consumers out of PL/SQL Profiler DBMS_PROFILER data." Oracle Support note 243755.1, 2012.

Oracle Corporation, "SQLT (SQLTXPLAIN) - Tool that helps to diagnose a SQL statement performing poorly or one that produces wrong results." Oracle Support note 215187.1, 2013.

Oracle Corporation, "Table Prefetching causes intermittent Wrong Results in 9iR2, 10gR1, and 10gR2." Oracle Support note 406966.1, 2007.

Oracle Corporation, "A Technical Overview of the Oracle Exadata Database Machine and Exadata Storage Server." Oracle White Paper, 2012.

Oracle Corporation, "TRCANLZR (TRCA): SQL_TRACE/Event 10046 Trace File Analyzer - Tool for Interpreting Raw SQL Traces." Oracle Support note 224270.1, 2012.

Oracle Corporation, "Understanding Bitmap Indexes Growth while Performing DML operations on the Table." Oracle Support note 260330.1, 2004.

Oracle Corporation, Oracle Database Documentation, 10g Release 2.

Oracle Corporation, Oracle Database Documentation, 11g Release 1.

Oracle Corporation, Oracle Database Documentation, 11g Release 2.

Oracle Corporation, Oracle Database Documentation, 12c Release 1.

Oracle Corporation, *Oracle Database 10g: Performance Tuning*, Oracle University, 2006.

Oracle Corporation, "Query Optimization in Oracle Database 10g Release 2." Oracle White Paper, 2005.

Oracle Corporation, "SQL Plan Management in Oracle Database 11g." Oracle White Paper, 2007.

Oracle Corporation, "Optimizer with Oracle Database 12c." Oracle White Paper, 2013.

Oracle Corporation, "SQL Plan Management with Oracle Database 12c." Oracle White Paper, 2013.

Oracle Corporation, "Use Caution if Changing the OPTIMIZER_FEATURES_ENABLE Parameter After an Upgrade." Oracle Support note 1362332.1, 2013.

Oracle Optimizer Blog (http://blogs.oracle.com/optimizer).

Osborne, Kerry, Kerry Osborne's Oracle Blog (http://kerryosborne.oracle-guy.com/)

Pachot, Franck, "Interpreting AWR Report – Straight to the Goal", 2014.

PHP OCI8, *PHP Manual*, 2013 (http://php.net/manual/en/book.oci8.php).

Põder, Tanel, Tanel Poder's Blog (http://blog.tanelpoder.com).

Senegacnik, Joze, "Advanced Management of Working Areas in Oracle 9i/10g." Collaborate, 2006.

Senegacnik, Joze, "How Not to Create a Table." Miracle Database Forum, 2006.

Shaft, Uri and John Beresniewicz, "ASH Architecture and Usage." Miracle Oracle Open World, 2012.

Shee, Richmond, "If Your Memory Serves You Right." IOUG Live! Conference, 2004.

Shee, Richmond, Kirtikumar Deshpande and K Gopalakrishnan, *Oracle Wait Interface: A Pratical Guide to Performance Diagnostics & Tuning*. McGraw-Hill/Osborne, 2004.

Shirazi, Jack, *Java Performance Tuning*. O'Reilly, 2003.

The Data Warehouse Insider Blog (https://blogs.oracle.com/datawarehousing).

Vargas, Alejandro, "10g Questions and Answers." 2007.

Wikipedia encyclopedia (http://www.wikipedia.org).

Williams, Mark, *Pro .NET Oracle Programming*. Apress, 2005.

Williams, Mark, "Improve ODP.NET Performance." *Oracle Magazine*, 2006.

Winand, Markus, *SQL Performance Explained*. 2012.

Wood, Graham, "Sifting through the ASHes." Oracle Corporation, 2005.

Wustenhoff, Edward, *Service Level Agreement in the Data Center*. Sun BluePrints, 2002.

Zait, Mohamed, "Oracle10g SQL Optimization." Trivadis CBO Days, 2006.

Zait, Mohamed, "The Oracle Optimizer: An Introspection." Trivadis CBO Days, 2012.

“互联网上充斥着大量的Oracle性能相关信息，不但高度碎片化，而且有很多是错误的。本书则异常清晰地给出了Oracle性能相关的理论和实践，明确指导读者找到需要达成的目的以及如何达成目的。”

——Jonathan Lewis，世界级Oracle技术专家，
英国Oracle用户组总监，《Oracle核心技术》作者

“这是一本技术与理念并重的参考书，不仅包含了大量完备的可重用的实例，而且包含了一些富有说服力的新观点。我可以用他的观点去说服更多的人做正确的事。”

——Cary Millsap，Method R公司首席执行官，
Oracle公司系统性能集团前副总裁，数据库性能技术大师

前端业务应用炙手可热之日，便是优化后端数据库性能之时。当此之际，身怀数据库优化绝技，可以让你平步职场，傲视群英。

本书是Oracle数据库优化专家Christian Antognini的一部继往开来的里程碑式著作。书中的最佳实践和诸多建议全部来源于作者在实战一线的丰富积累，不仅简单实用，而且发人深省，堪称一座“宝库”，适合各层次读者研读和发掘。

与其他同类图书不同，本书不仅涵盖了当前可用的各种Oracle版本，还指明了各个版本独有的性能优化特性。全书以崭新的视角开篇立论，围绕查明问题真相和搜寻有效方略，透彻讲解了查询优化器的配置，表访问、连接和物理表布局的优化，以及加速SQL执行计划等重要主题，被读者誉为“最透彻，但又最通俗的性能优化好书”。

与本书第一版相比，作者增加了关于Oracle Database 11*g*和Oracle Database 12*c*的内容，补充了层次剖析工具、ASH、AWR和Statspack等知识点，并根据可读性重新组织了部分素材。

Apress®

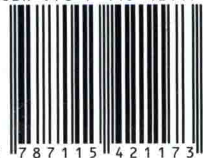
图灵社区：iTuring.cn

热线：(010)51095186转600

分类建议 计算机／数据库／Oracle

人民邮电出版社网址：www.ptpress.com.cn





ISBN 978-7-115-42117-3

定价：119.00元

[General Information]
书名=图灵程序设计丛书 ORACLE性能诊断艺术 第2版
作者=（瑞士）CHRISRIAN ANTOGNINI著
页数=610
SS号=13967414
DX号=
出版日期=2016.06
出版社=北京人民邮电出版社